APPROVAL SHEET

Title of Thesis: The Lightweight Virtual File System

Name of Candidate: Navid Golpayegani
Ph.D. in Computer Science,
2017

Thesis and Abstract Approved: _____

Dr. Milton Halem
Research Professor
Department of Computer Science and
Electrical Engineering

Date Approved: _____

The Lightweight Virtual File System

by
Navid Golpayegani

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Ph.D. in Computer Science
2017

ACKNOWLEDGMENTS

I would like to thank Dr. Milton Halem, Dr. Yelena Yesha, and Dr. John Dorband from University of Maryland Baltimore County for their help and support in my graduate career. I'd also like to thank Dr. Curt Tilmes, and Edward Masuoka from NASA Goddard Space Flight Center for their advice throughout my graduate school years and the flexibility they provided me to work on my graduate school research. Additionally, I'd like to thank Bryan Wyatt, Robert Warmka, Jon Trantham, and Chris Markey from Seagate for providing me with hardware and software support. I'd like to thank Damon Earp and Jihad Ashkar from SSAI for their help with this project. I would also like to thank NASA Goddard Space Flight Center and the Center for Hybrid Multicore Productivity Research for providing me with the necessary infrastructure to perform my work. Finally, I'd like to thank my family for their encouragement to help me get to this point.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CODE LISTINGS

Chapter 1

INTRODUCTION

1.1 Background

Data storage, retrieval, and organization is the backbone of any compute system ranging from the most powerful supercomputers to the latest cloud computing providers. As new and more complex data mining and analytic systems are developed, there is a growing need for storing larger amounts of data and maintaining historical data for longer time periods. This creates a unique problem to create a consistent data structure for users while adding newer storage hardware to existing ones. Most existing file systems are not developed in a platform agnostic way such that they can run on new storage architectures as well as decades old storage architectures.

In a recent study by the International Data Corporation (IDC), it is estimated that generated data volumes will double every two years between 2012 and 2020 (Gantz & Reinsel 2012). his includes data in all sectors like telecommunication, entertainment, science, etc. This growth can be seen in Figure 1.1.

In the academic sector, the growth of data from the science disciplines is fueled by better sensors that take measurements with more spatial and temporal resolution and with finer spectral resolution. For example, the Large Hadron Collider (LCH), a particle collider in Europe, produces 19 gigabytes of data per minute. In 2010 alone, the LHC created 13 petabytes of data (Brumfiel 2011). Similarly, the Sloan Digital Sky Survey (SDSS), stores more than 140 terabytes of data since 2000 at a rate of about 200 gigabytes per night. However, the successor to SDDS, the Large Synoptic Survey Telescope, expected to become operational in 2016, will produce that amount of data in five days (The Economist 2014). Figure 1.2 shows the the National Oceanographic and Atmospheric Administra-

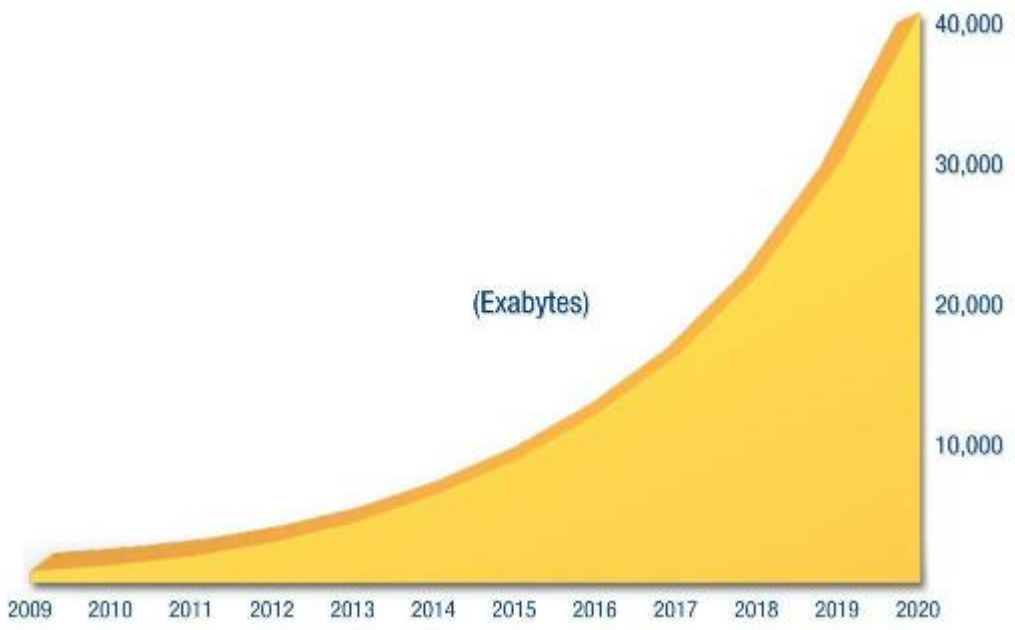


Figure 1.1: IDC Digital Universe study showing a 50 fold growth of data from 2010 to 2020 (Gantz & Reinsel 2012)

tion's (NOAA) NEXRAD archive size since 1991. We can see that until approximately 2008 the archive size was growing at a linear rate. Since 2008, however, the NEXRAD archive size has been increasing at a higher rate.

In addition to better sensors contributing to the growth of data, there's a increasing demand for long-term data archives. These long-term archives help support decadal studies in many scientific fields. For example in (Jeger & Pautasso 2008) show a survey of research papers which uncovered links between global change and plant disease which were impossible to do without long-term datasets. Similarly, in (Tommasi *et al.* 2017) they show the importance of seasonal and decadal climate forecasts in how they relate to marine biology.

The primary resource for managing the data are the data center's file systems. The primary role of a file system is to perform two main tasks: metadata management and data management. Metadata management involves the organization of data which is usually represented as a directory structure. It also includes other information such as the permissions of files and directories and other similar metadata information. The data management task is responsible for the actual storage of the content.

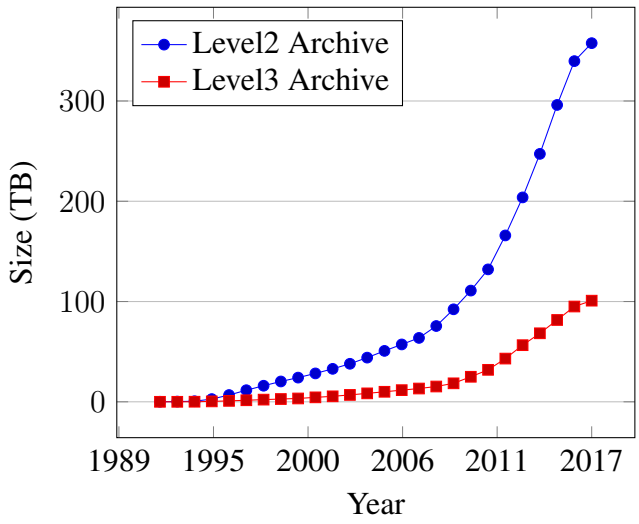


Figure 1.2: NEXRAD Archive Size showing non-linear data growth for the National Climatic Data Center at National Oceanographic and Atmospheric Administration. Retrieved from (National Oceanographic and Atmospheric Administration 2014) on June 24th, 2017

1.2 Motivation

Storing such high volumes of data and managing these datasets over decades creates unique challenges for data centers and the file systems they use. High volumes of data require a large number of physical drives which, in turn, requires managing multiple file system instances or file systems capable of spanning multiple physical drives. Supporting projects needing to use datasets over decades creates additional challenges to maintain diverse hardware and software technologies. Over time the storage industry has diversified to include solid state disks and object based storage in addition to block based disks. Similarly, software to support a storage cluster have evolved to encompass systems such as Network File System (NFS) (Shepler *et al.* 2003), Simple Storage System (S3) (Amazon Web Services, Inc. 2013), and Kinetic Storage (Seagate Technology LLC 2017).

Figure 1.3 shows a common layout currently used in data centers. A common approach in data centers is to assign different nodes to different tasks such as public servers, responsible for interacting over the internet, compute nodes, responsible for producing new data, a file system, responsible for storing all the generated data from the compute nodes, and a project specific metadata server, responsible for storing very detailed metadata information about the generated data.

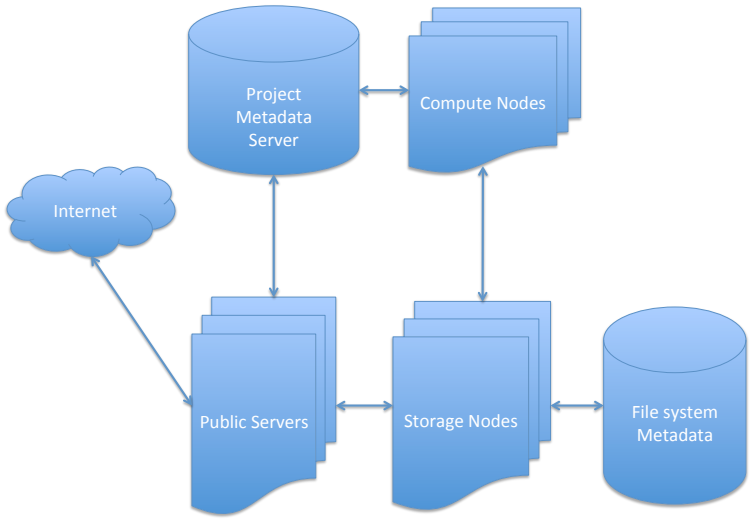


Figure 1.3: Common layout of a data center showing compute, processing, and storage nodes

The combination of large data volumes, long-term dataset support, and data center design create scalability and long term issues. Listing 1.1 shows a real world example of a computer in a data center having access to small number of disks. Lines 1–5 show five local disks being mounted on the computer. Lines 7–23 show several remote disks from two computers mounted on the computer. Finally, lines 25–29 shows existing paths being bind mounted as other paths. A bind mount is copying the directory structure of one location to another location. In addition to the mounting structure, listing 1.2 shows the contents of a single directory under /MODAPSint on the same computer. Paths in that directory consist of symbolic links to other paths pointing to mount points on other remote computers.

This example illustrates how the storage architecture of a data center can become more complex over time. This complexity stems from several factors: (1) the desire to create a uniform layout to data consumers while (2) continuing to support data growth over several decades, (3) addition of new projects while maintaining older projects and (4) the introduction of new hardware and software technologies.

A data center might use the most modern and stable storage architecture at the start

```
1  LABEL=f0                     /archive     xfs   defaults        0 0
2  LABEL=f1                     /archive/f1  xfs   defaults        0 0
3  LABEL=f2                     /archive/f2  xfs   defaults        0 0
4  LABEL=f3                     /archive/f3  xfs   defaults        0 0
5  LABEL=f4                     /archive/f4  xfs   defaults        0 0
6
7  fs4:/cm                      /cm          nfs   tcp,rw          0 0
8  fs4:/modular_sstg/newstig    /newstig     nfs   tcp,rw          0 0
9  fs4:/modular_sstg/sstg1      /sstg1       nfs   tcp,rw          0 0
10 fs4:/modular_sstg/stigdev3   /stigdev3    nfs   tcp,rw          0 0
11 fs4:/modular_misc/data1      /data1       nfs   tcp,rw          0 0
12 fs4:/modular_misc/geocp      /geocp       nfs   tcp,rw          0 0
13 fs4:/modular_misc/modisnfs1  /modisnfs1   nfs   tcp,rw          0 0
14 fs4:/modular_misc/cmgroup    /cmgroup     nfs   tcp,rw,bg,intr  0 0
15 fs4:/modular_home            /home        nfs   tcp,rw          0 0
16 dev1:/MODAPSint              /MODAPSint   nfs   tcp,ro,bg,intr  0 0
17
18 fs4:/mnt/SSTG2               /SSTG2       nfs   tcp,rw,bg,intr  0 0
19 fs4:/SSTG/SSTG3              /SSTG3       nfs   intr,bg,rw,tcp  0 0
20
21 fs4:/mnt/modular_test        /test        nfs   tcp,rw,bg,intr  0 0
22 fs4:/mnt/raid1               /raid1       nfs   tcp,rw,bg,intr  0 0
23 fs4:/mnt/raid3               /raid3       nfs   tcp,rw,bg,intr  0 0
24
25 /raid1/test2                 /test2       none  bind
26 /raid1/L1A                   /L1A         none  bind
27 /cm/cc                       /cc          none  bind
28 /ftp/cmgroup                 /cmgroup/ftp none  bind
29 /ftp/mtvs                    /mtvsraid1   none  bind
```

Listing 1.1: Real World fstab file mounting local and remote disks

```
1  2007142062150.hdf -> /SSTG/UTdata/LAND/MCD43B1/2007142062150.hdf
2  2007141165711.hdf -> /SSTG/UTdata/LAND/MCD43B1/2007141165711.hdf
3  2007141141920.hdf -> /SSTG/UTdata/LAND/MCD43B1/2007141141920.hdf
4  2007142175119.hdf -> /SSTG/UTdata/LAND/MCD43B1/2007142175119.hdf
5  2007143103256.hdf -> /SSTG/UTdata/LAND/MCD43B1/2007143103256.hdf
6  2007142093542.hdf -> /SSTG/UTdata/LAND/MCD43B1/2007142093542.hdf
7  2007142105359.hdf -> /SSTG/UTdata/LAND/MCD43B1/2007142105359.hdf
8  2007140232854.hdf -> /SSTG/UTdata/LAND/MCD43B1/2007140232854.hdf
9  2007143032329.hdf -> /SSTG/UTdata/LAND/MCD43B1/2007143032329.hdf
10 2007143092747.hdf -> /SSTG/UTdata/LAND/MCD43B1/2007143092747.hdf
11 2007142102318.hdf -> /SSTG/UTdata/LAND/MCD43B1/2007142102318.hdf
12 2007143011024.hdf -> /SSTG/UTdata/LAND/MCD43B1/2007143011024.hdf
13 2007141165657.hdf -> /SSTG/UTdata/LAND/MCD43B1/2007141165657.hdf
14 2007143074815.hdf -> /SSTG/UTdata/LAND/MCD43B1/2007143074815.hdf
15 2007143034708.hdf -> /SSTG/UTdata/LAND/MCD43B1/2007143034708.hdf
16 2007143031546.hdf -> /SSTG/UTdata/LAND/MCD43B1/2007143031546.hdf
17 2007142110656.hdf -> /SSTG/UTdata/LAND/MCD43B1/2007142110656.hdf
18 2007142015350.hdf -> /SSTG/UTdata/LAND/MCD43B1/2007142015350.hdf
```

Listing 1.2: Sample Directory from one of the mountpoints in listing 1.1

of a project. The most up to date technology used original for the above example was to use individual disks with NFS mounts. Due to slow network access at the time some, more commonly used data, was stored on local drives. While adding more local and remote disks might be a convenient way for the data center to grow the storage architecture this creates a major inconvenience for the projects. Data producers for the project would have to continually maintain a table to identify the mapping of individual files to local/remote disks. In the above example this was accomplished via symbolic links and bind mounts.

Changing requirements, adding features, and supporting new projects exacerbate such problems. Different projects or new features might require datasets to be organized in a different layout than the physical one provided by the data center or the virtual one created by the symbolic links and bind mounts for the original project. To accommodate the new layout more symbolic links and bind mounts might be added.

Finally, adding new hardware or software technologies might be difficult to do to existing projects. While not visible in the above example, the addition of new storage technologies such as HGST's Active Archive System (AAS) (Western Digital Corporation 2017), DDN's WOS (DataDirect Networks 2017), or Seagate's Kinetic Drives creates complex problems. These storage architectures do not support standard the POSIX file

standard (The Open Group 2017) so are not able to be integrated with legacy system and require custom software for interaction. To support such systems the projects using the data center's resources require custom code and detailed knowledge of each storage system to support. This creates a heavy burden on the project developers to learn and understand each non-POSIX system.

A common approach to addressing some of these issues is through the use of a metadata server such as a database system. A project can store rich metadata information in a metadata server along with the location of the data store. When a computation needs to be performed on the data, the metadata server is queried for the location of the file and then retrieved. This alleviates some of the problems by reducing the number of symbolic links or bind mounts. However, this does not completely eliminate their need. While custom applications developed by the project can be programmed with knowledge of the metadata server, most third party applications would not work out of the box. An example would be simple data distribution such as via FTP or HTTP.

In addition to not entirely eliminating the symbolic link problems this setup creates a synchronization problem between the metadata server and the storage architecture. With large data centers hardware failures are unavoidable. Most data centers will have redundancy and will regularly move files two new drives. Additionally, projects rarely generate data only once. Mistakes in algorithms or improvements in algorithms usually result in data being replaced. These constant changes create a synchronization problem when trying to ensure the information in the metadata server is consistent with the information on the drives.

Our approach is to develop a new kind of storage architecture tailored for such data centers. This new storage system will be capable of supporting petabyte and exabyte scale data sets while allowing for new file system features such as completely separating data center file allocation from project file organization. Additionally, we ensure that our approach is lightweight enough to run on commodity hardware without the need of high amounts of memory or local disk storage. We address the above described shortcomings in current file systems with LVFS.

We use the MODIS Adaptive Processing System (MODAPS) as a use case for our approach. MODAPS is part of NASA's Earth Observing System Data and Information System (EOSDIS). MODAPS produces approximate 760 gigabytes of data per day ranging from data directly received from the satellite to gridded data for several instruments in

operations since 1999. Since then, MODAPS has produced over 30 petabytes of data in over 2 billion files. These files are distributed to the public via FTP, HTTP, OPeNDAP, and other protocols. The distribution of data via FTP accounts for 90 Terabytes per month and averages approximately 300 users per day with a peak of 600 users per day.

Chapter 2

THESIS STATEMENT

Developed, implemented, and tested a Lightweight Virtual File System (LVFS) and a framework consisting of a variable set of functional plugins, which enable the capabilities to dynamically rearrange directory tree structures, streaming format conversions, and merging of incompatible storage architectures to form an integrated system applicable to current and future storage technologies

LVFS separates standard file system functions, such as data storage, data location, and data organization into loosely coupled components, which are chained together to create a dynamic, flexible, platform agnostic, lightweight virtual file system. LVFS is a full featured self contained file storage system.

LVFS is easy to use, deploy, and is open source.

2.1 Problem Definition

A file system has two major components: storage and metadata. The storage component is responsible for determining how to store the data on the physical hardware including any allocation or necessary redundancy computations to be made. The metadata component is responsible for storing the organization of the data and any access control information. The data organization is usually displayed as a directory structure. These components are usually developed as a software package that is tightly coupled and inaccessible to anybody except to the file system developers.

The storage component of a file system is responsible for the distribution of the data

stored in the file system. The storage component needs to tackle several problems including the distribution of data among storage nodes and the maintenance of high throughput of data to/from the storage nodes. This component has been a well studied subject and many algorithms exist to attempt to solve this problem (Honicky & Miller 2004; Shreedhar & Varghese 1995; Azar *et al.* 1999). For example, a round robin distribution of data results in an easy and fast algorithm. However, if storage nodes have varying capacities, this algorithm will result in the uneven distribution of data across all storage nodes thereby exhausting the available space on the smaller storage nodes first. Additionally, due to the predictable nature of the round robin algorithm, patterns can develop in the way data is distributed which could create bottlenecks. A random distribution of data would help with the bottleneck issues of the round robin method, however, such an algorithm requires additional information to maintain the location of data. While the additional information could be small, when dealing with large number of files, even such small amounts add up to significant amount of additional information to be maintained and stored.

The metadata component of a file system faces unique challenges in number of operations it can support, the abilities it provides to search and locate data stored in the file system. Frequently file systems will create a monolithic system which contains both the data and metadata components. However, such a design can result in the metadata component becoming a bottleneck. This is due to every file system operation needing to contact the metadata component first. To address this possible bottleneck some file systems separate the metadata and data components into two tightly coupled independent systems. With this design after initially contacting the metadata component for the necessary information, all file operations only need to communicate with the data component. This separations improves metadata performance as file reads and writes no longer need to access the metadata component. Even with such improvements however, the metadata components face the challenge of providing a fast way for data to be discovered in Big Data systems where billions of files are stored.

2.2 Contribution

Plugin Framework

LVFS consists of a unique storage framework concept based on variable plugins which interact with each other through the LVFS core code. With this framework

design, each plugin is assigned a single, well defined function. When chained together by LVFS, these plugins create full featured file system capable of handling current and future storage architectures.

LVFS' plugin framework is divided into several categories:

- Cache
- Content
- Database
- Directory
- Filter

Each plugin category defines the tasks. For example, *Directory* plugins are responsible for generating file or directory listings and provide basic information such as file type, file size, etc. *Content* plugins are responsible for retrieving or storing file content.

We ensure that this plugin framework design can handle any standard file system task by creating a 1:1 mapping of required POSIX file system functionality to a function in one of the above plugin categories.

Replacement of File System Metadata Component

LVFS removes the file system metadata component entirely from its software stack. This unique approach increases scalability, simplifies the code, reduces synchronization issues between the file system and the project's metadata server, and removes duplication of information. This contribution may be the first such approach to file system design.

Dynamic Views

LVFS is capable of dynamically reorganizing the directory structure to ensure a convenient design for both data center designers and project users. Since LVFS relies on the rich metadata information provided by the project's metadata server, it is capable of dynamically re-arranging the same data into different layouts. This unique feature of LVFS creates a powerful ability for projects. For example, a remote sensing project would be capable of making the same file available organized by different attributes such as measurement time or measurement location.

Traditional file systems would accomplish such a task by creating a directory structure to store files organized by measurement time. To make the same file available by measurement location using a traditional file system, the project could create another directory structure for location based searches and then provide symbolic links to the same files in the directory structured by measurement time. While possible for small projects such a solution would not be scalable. For example, our case study, MODAPS, which maintains 2.5 billion files would require to maintain an additional 2.5 billion symbolic links to maintain a directory structure for searching by each new attribute.

With LVFS no symbolic links are necessary. The addition of a few lines in the LVFS configuration file for each new directory structure would allow LVFS to dynamically create each new structure on demand.

Flexible file content

Metadata is not the only aspect of LVFS that is controlled via plugins. File content in LVFS is another. This unique way of managing file content gives LVFS the level of flexibility to deal with any kind of hardware or software architecture. Over the decades as storage architecture changes, a data center can continue to support the old architecture and the new one simply by loading appropriate plugins into LVFS for both architectures. Examples of plugins already provided by LVFS include plugins which retrieve file content from standard block based disks, to plugins for object based storage, and from database systems such as PostgreSQL (The PostgreSQL Global Development Group 2017).

LVFS takes this concept one step further by allowing concatenation of content for varying sources. With a configuration setting in LVFS, it can be configured to obtain the file content via multiple sources. While the use cases for such a feature are numerous, a common one could be for space saving purposes. A data center providing billions of files all with a common header could save significant storage space by only having the common header stored in a single location. With the appropriate configuration, LVFS can append this common header to all other files provided by LVFS. This creates the illusion that all files contain this header.

Redundant file content

Similar to the flexible file content, LVFS is capable of utilizing redundant file content. With this feature alternate locations can be specified in case the primary location is unavailable. While such a feature is available in many other file system architectures, LVFS provides a unique twist to this feature. LVFS is capable of using different storage architectures for the redundant location. For example, with LVFS, the primary location of a file could be on a standard block based disk while a secondary instance of the file is located on an object based storage system like the Seagate Kinetic drive. LVFS will provide the file to users from either location.

Streaming content verification

When dealing with petabytes of data in billions of files most files being accessed will be retrieved from remote sources. This means that usually some sort of IP protocol will be employed, such as TCP or UDP. As discussed in (Stone & Partridge 2000) somewhere between 1 in 16 million to 1 in 10 billion packet checksums fail to detect data corruption. Assuming the best case scenario of 1 in 10 billion failure and assuming all TCP packets are sent using the theoretical maximum of 64 Kilobytes this means there will be an undiscovered file corruption every 640 Terabytes. For example, the MODAPS system distributes 1.3 Petabytes of data between storage and compute nodes every month. This means that every month at least 2 files have corruption which go undetected.

To combat this problem, LVFS incorporates what we call streaming content verification, when files are stored in LVFS, an checksum is computed for each file. This checksum is stored along with the file and is used for verification to ensure no corruption has happened during transmission or while the file was at rest on the storage node.

On-Drive MapReduce computation

The flexibility of LVFS extends to supporting computations on-drive. LVFS can be configured with a plugin which supports the MapReduce programming model (Dean & Ghemawat 2004; Zhao & Pjesivac-Grbovic 2009). With this plugin and the necessary hardware, such as Seagate's Active Drives, LVFS is capable or performing on-drive computation. This significantly reduces the amount of data that leaves the drives. Using this plugin, users are capable of submitting a Map function to be registered with each supported drive. An accompanying Reduce application is made

available on the local computer. The Reduce application executes on the host computer with a list of files. Those files are passed to LVFS which determines the Active drives on which they reside. The Map function is executed on those drives and the results from the Map function are returned to the Reduce application for final processing.

Streaming Format Conversions

Another unique contribution of LVFS is streaming file format conversions. This feature addresses a common problem of long running projects. During the initial inception of the project decisions are made to store data in a file format suitable at the time. For long running projects, the file format choice made decades ago are likely no longer good choices now. A project will have several hard choices to make in terms of changing file formats:

1. Continue with old file format
2. Update file format for newly generated files only
3. Update file format for new files and convert old files

All options have undesirable consequences. Option 1 could result in the community not using the project's data anymore. As file formats change, tools capable of handling old file formats become harder to find and users might abandon using the datasets. Option 2 would create difficulties when comparing long term records due to format changes. Option 3 would create difficulties for the data center since they would require additional processing power to convert old formats to new ones and extra storage to temporarily store both formats while transitioning. Option 3 would also have scalability issues due to the increasing storage and processing requirements as projects run for longer times.

LVFS provides the unique ability to perform on the fly data conversions. The plugin architecture of LVFS allows a data center or project to specify a plugin which performs the format conversion. To the end users it appears as if the same dataset is available in multiple formats but in reality, LVFS will convert from one format to another as files are accessed. This feature allows a project or data center to maintain the old file formats while using a plugin to convert them to the latest file format. No ad-

ditional disk space is required and the only CPU power needed is for the on-demand conversion.

Hybrid Decentralization

Since LVFS does not have an internal metadata component, it can't make any assumption about the stability of the project's metadata server. LVFS is built with the assumption that the project metadata server can be unavailable for prolonged periods. In addition to being built with extensive error checking to ensure proper functionality under many different error conditions, LVFS is capable of continuing execution without the availability of a metadata server. LVFS is capable of running in decentralized mode as both a primary or backup mode of operation.

Decentralized mode in LVFS has several advantages and some disadvantages. The primary advantage is the removal of the only single point of failure in LVFS, the project's metadata server. In addition to being the single point of failure, the metadata server is frequently also the bottleneck in many file systems. LVFS generates directory structure from information retrieved via the project's metadata server. An outage of that server will result in LVFS not displaying a directory structure. However, under certain configuration choices, LVFS will be able to continue functioning even without a directory structure. For most data centers and projects the directory layout and file naming conventions are known very well internally. If a user knows which file they wish to access they will know the path under which the file is available and what the naming convention should be. Providing the full path to the file, LVFS will be able to return the file without having access to the metadata server.

Lightweight

LVFS is built on top of existing and stable software stack. As a result, LVFS' code base is small and concise. It is written mostly in C++ with a little bit of C and python code where necessary. LVFS is able to run on computers as powerful as the latest models with hundreds of gigabytes of memory and terabytes of local disk storage to computers with only a few hundred megabytes of memory and nearly no disk space availability.

LVFS plugin architecture helps with the lightweight design by allowing new features to be added into LVFS with minimal amount of code. A typical LVFS plugin

performs a single task and is combined with other plugins to create the whole file system experience. Each plugin, therefore, is lightweight consisting, on average, of only 130 lines of C++ code. Even though LVFS is both multi-threaded and event based most plugins are sufficiently protected such that they do not have to manage any concurrency problems that arise from being multi-threaded.

Benchmarking

LVFS provides extensive benchmarking support for Ganglia (Massie, Chun, & Culler 2004) and collectd (Forster 2017). Once again, the plugin nature of LVFS allows it to be expanded to other benchmarking tools. Not only is the core LVFS code capable of providing benchmark statistics so are any plugins either as part of LVFS or third party provided ones. With this design data centers and projects can easily monitor the performance of LVFS and identify any bottlenecks for either code improvements in LVFS or configuration changes at the data center. While LVFS only provides the raw benchmarking numbers, the plugins will feed those numbers to systems which provide easy to view graphs. Ganglia has built in visualization while colletd users will likely use systems such as Grafana (Grafana Labs 2017).

The core LVFS code provides statistics such as how many POSIX requests it has handled at any given time. This is convenient for calculating how many I/O operations per second LVFS is capable of performing. Each plugin can provide any arbitrary amount of benchmarking statistics. For example, a metadata plugin, which might use PostgreSQL (The PostgreSQL Global Development Group 2017), could provide information on how many database connections it keeps open, how much memory it uses, and how many database queries it performs. This information will help the project and the data center determine if their database schema or the queries they use to provide LVFS with the necessary metadata information needs improving.

Configuration

Unlike standard file systems, LVFS behavior and directory structure is controlled via a human readable and modifiable configuration file. This configuration file performs similar functionality to other file system's superblock and inode structure. With most file systems the superblock and inode structure is controlled via file system utilities. Some changes, especially to the superblock, sometimes can only be performed destructively by reformatting a drive.

LVFS, on the other hand, encourages the file system and directory tree structure to be changed dynamically. With this design LVFS can be reconfigured dynamically without the need to migrate data to make changes to the file system. It also creates a storage system designed to adjust to meet the user's needs rather than forcing the user to adjust to the storage system.

LVFS Community

While a lot of functionality is provided by LVFS directly it is nearly impossible to provide plugins for every foreseeable technology. The plugin capability of LVFS allows it to create a community of developers which can contribute new plugins to LVFS and extend it to new technologies existing now and in the future. The goal of LVFS is create a community repository of plugins which users can use to obtain new plugins or add their own plugins to share with the rest of the community.

Chapter 3

RELATED WORK

Implementing file systems for large scale use in data centers poses challenges including those of reliability, availability, expandability, speed, and organization. File systems commonly only address a subset of these challenges. Throughput is a common problem solved by most file system. The solutions to the problems of reliability and availability tend to have common roots and are frequently addressed together. The problems of expandability and data organization are rarely addressed in file system development. Expandability problems are those dealing with the long term storage issues such as the changing file formats or introduction of new storage technologies we discussed in section 2.2. Data organizations are the issues we discussed in the same section pertaining to the differing requirements for directory layout for the same dataset. In this section we look at how different file systems attempt to solve some of these issues.

3.1 Hadoop Distributed File System

The Hadoop Distributed File System's (HDFS) (Borthakur 2008) was developed by the Apache group to primarily to be used with Apache Hadoop (White 2009). HDFS makes several assumptions including:

- Hardware failures are the norm
- Stored datasets are gigabyte or terabyte sized
- It is cheaper to move computation to data

HDFS is designed as a block based file system. Files stored in HDFS are split into fixed sized blocks and stored on storage nodes, called DataNodes. The metadata server,

called the NameNode, is responsible for managing the mapping of files to data blocks. To access a file, a client first contacts the NameNode to determine which DataNodes store the blocks for the desired file. Once a list has been compiled, the client contacts DataNodes directly for access to the blocks. Each file is assigned a replication factor which determines how many copies of the file are stored on the DataNodes. The direct communication of clients to DataNodes allows for high throughput and the data replication insulates HDFS against node failures. However, the use of a central authority for mapping files to blocks creates a bottleneck and single point of failure among other scalability problems (Shvachko 2010).

Compared to HDFS, LVFS makes several different choices. A primary difference between LVFS and HDFS is that LVFS removes the assumption that the system will store few large files. Therefore, LVFS does not take a block based approach to storage. With LVFS, files are stored in their entirety as a single object similar to other object based storage systems. Additionally, LVFS does not have its own internal metadata server. Instead, LVFS relies on a project's specific metadata server to obtain its information for directory listing and, optionally, for file location. The advantage to this design is that, unlike HDFS, LVFS is capable of not having a central point of failure and bottleneck similar to HDFS' NameNode.

3.2 Lustre

The Lustre file system (Schwan 2003) consists of three major components: the Metadata Server (MDS), the Object Storage Server (OSS), and the clients. The MDS consists of one or more Metadata Targets (MDT). Each MDT stores metadata information such as directory structure, file name, permissions, etc. The OSS consists of one or more Object Storage Targets (OST). Each OST represents a physical disc used for storage. Similar to HDFS, once files are located in the MDS, all file I/O operations do not involve the MDS.

While Lustre addresses the scalability problems of the MDS by distributing the contents among multiple MDTs, Lustre still requires a strictly controlled metadata server. The information stored in the MDS is pre-defined by Lustre. Due to this tight control, Lustre only follows the standard static directory layout like most other POSIX compliant file systems. LVFS, however, does not rely on an internal metadata server. With information provided in the LVFS configuration, it is capable of creating dynamic views of the same data sets without the need of any duplication or unnecessary information. To achieve a sim-

ilar kind of functionality in Lustre, symbolic links will have to be added creating additional data which needs to be stored and managed.

3.3 Parallel Virtual File System

Just like the file systems described above, the Parallel Virtual File System (PVFS) (Ross & Thakur 2000) consists of several components including a metadata component and an I/O component. Similarly, PVFS avoids the use of the metadata component during the I/O stage to avoid performance problems. PVFS primarily attempts to deal with the scalability problem of I/O throughput but does not deal with the scalability issues of managing large number of files with a few small exceptions (Vilayannur *et al.* 2002).

Similar to the file systems above, PVFS has an internal metadata server that is tightly controlled by the file system. As a result, only information allowed by PVFS is stored in the metadata server. As mentioned above, this design results in the drawback that symbolic links have to be used to create multiple views of the same data set which LVFS addresses by removing the metadata server and relying on the project metadata server.

3.4 High Performance Storage System

The High Performance Storage System (HPSS) was developed by IBM (Watson & Coyne 1995) for mainly dealing with parallel I/O operations. HPSS relies on the Distributed Computing Environment (Group 2014) to achieve parallelism. Unlike storage systems mentioned above, HPSS relies on a DB2 (Karlsson *et al.* 2001) database for metadata storage. This design is similar to LVFS' design. However, unlike LVFS the schema used by HPSS in the database is controlled by the file system. This means that metadata information stored in the database is duplicated between the file system metadata and the project's metadata systems creating a synchronization problem. Additionally, even though the file system relies on a database for metadata information, it is not capable of providing dynamic views by retrieving the information from the database differently. This results in the use of symbolic links for the simulation of the dynamic views feature provided by LVFS. Similar to the file systems above, HPSS does not have the flexibility provided by LVFS with its plugin architecture.

3.5 SMART-IO

Unlike the above mentioned file systems, the SMART-IO file system (Tian *et al.* 2012) attempts to specialize for scientific datasets only. SMART-IO uses dynamic organization of multidimensional datasets to speed up read throughput. Like most other file systems described above, SMART-IO's goal is improved I/O throughput. Problems related to data discovery when dealing with large scale file systems are not addressed by SMART-IO.

3.6 Amazon S3

Amazon's Simple Storage System (S3) (Amazon Web Services, Inc. 2013) is advertised as a decentralized storage system. S3 is an implementation of an Object-Based storage system (Mesnier, Ganger, & Riedel 2003). While it is fully decentralized, and therefore does not suffer from any single point of failure, it is limited to a very strict flat structure. All S3 based objects are stored organized by a unique user assigned key. While this design allows for fast access to existing file it makes data discovery nearly impossible. Additionally, S3 is not POSIX compliant so any tools not specifically programmed for S3 are not capable of accessing data stored there without the use of third party software to act as a translation layer for the POSIX API to the S3 API.

3.7 Cassandra

Apache's Cassandra (Lakshman & Malik 2010) is another fully decentralized storage system. It is based on Amazon's Dynamo (DeCandia *et al.* 2007) storage system concept. Cassandra is usually considered a database rather than a file system while Dynamo is usually considered a key-value storage system. As a result these implementations are usually geared towards replacing database systems with little support for use as a file system.

3.8 Spyglass

Spyglass (Leung *et al.* 2009) attempts to exclusively improve the problem of data discovery for large scale file systems. Spyglass' approach is based on adding additional indexes on top of a file system. This extra index allows for faster querying of the existing metadata. Spyglass attempts to speed up queries such as finding all files with access time

larger than a value and a file size greater than some other value. Spyglass, however, only utilizes the existing metadata of a file system. Using only the existing metadata of the file system limits queries to only those based on the limited metadata server provided by the file system. Additionally, since Spyglass still requires the file system's metadata, an installation utilizing Spyglass will still suffer from synchronization problems resulting from duplicating metadata between the file system's metadata server and the project's metadata server.

3.9 Pantheon

Pantheon (Naps, Mokbel, & Du 2011) is a research project attempting to solve some of the same problems as LVFS using a different approach. Pantheon's approach is to adapt database concepts, such as query optimizations and B+ Trees, to create an additional layer on top of an existing file system directory structure. This extra layer can then be queried to search for files of interest. This approach is limited in that it can only rely on information stored in the file system. The very rich metadata information stored in the project's metadata server is never used. By relying on the rich metadata information from the project's metadata server and by relying on the projects database server storing the metadata, LVFS utilizes the same advantages as Pantheon but also has access to all the rich metadata information unavailable to Pantheon. In addition to the metadata access, LVFS having the ability to load custom plugins for most aspects of a file system creates flexibility not provided by Pantheon or other file systems described above

3.10 iRods

iRODS (Rajasekar *et al.* 2010) is another file system mainly concentrating on the challenges of data discovery in large scale file system. iRODS addresses these problems by utilizing a database server for metadata storage. Similar to LVFS, the file system layout is generated by querying the iRODS database. Unlike LVFS, however, the database and schema used for storing the metadata comprising the file system is dictated by iRODS. This means that project specific information can not be stored in iRODS which results in having to maintain a separate database for iRODS and one for the project resulting in the synchronization issues described earlier. iRODS also does not have the plugin capabilities

of LVFS so it is not capable of being expanded as hardware and software technologies change.

3.11 Recon

Similar to LVFS, the Recon system (Fryer *et al.* 2012) is an attempt at addressing some of the problems in the metadata component of storage systems. Specifically, Recon addresses the corruptions that can happen in the storage system's metadata component. Fryer et al. claim the increasing complexity of the metadata component is one of the causes of corruption. To address this, Recon uses additional information, called consistency invariants, that are checked at file system commit time to reduce the likelihood of errors. They provide a proof of concept implementation for the ext3 linux file system. While both Recon and LVFS focus on the metadata of a storage system, the LVFS approach is to reduce the complexity of the metadata component and, therefore, reduce the chance of metadata corruption.

3.12 SmartStore

SmartStore (Hua *et al.* 2009) attempts to deal with the unique big data challenges of data discovery by restructuring the internal metadata system of the file system. With SmartStore files are grouped and stored according to their metadata rather than the directory layout. Files are organized in a decentralized semantic fashion allowing for semantic analysis tools, such as Latent Semantic Indexing (LSI) (Papadimitriou *et al.* 2000). By combining physical data distribution with the logical semantic correlation SmartStore is scalable to exabyte systems. SmartStore relies on it's own internal methods for metadata distribution and, therefore, suffers from the same synchronization and metadata duplication problems as other systems described in this section as well as lacking the flexibility provided by LVFS' plugin architecture.

3.13 On-Drive Chromosomal Microarray Analysis

In bioinformatics similar technologies have been used by biostatisticians to perform Chromosomal Microarray Analysis (Delmerico *et al.* 2009) for cancer research. While technically not a file system, this experiment was to evaluate the viability of extracting

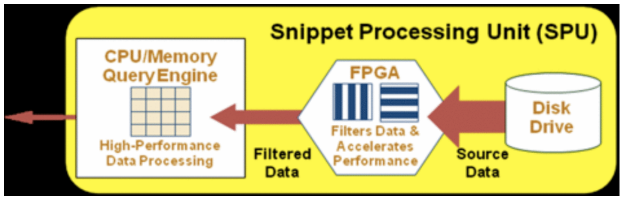


Figure 3.1: An SPU consisting of a Disk, FPGA, and general purpose processor (Delmerico *et al.* 2009)

highly correlated values from a dataset consisting of hundreds of gigabytes or more of double precision matrices. This experiment is similar to the LVFS contribution of performing on-drive computation. In this experiment the authors performed the correlation analysis using an active drive architecture (Acharya, Uysal, & Saltz 1998; Riedel, Gibson, & Faloutsos 1998; Keeton, Patterson, & Hellerstein 1998; Riedel *et al.* 2001; Riedel 1999). The active drive architecture developed consisted of a Snippet Processing Unit (SPU). Figure 3.1 shows the components of an SPU; a Disk, an FPGA, and a general purpose processor. The SPU cluster was developed as a database cluster in which the FPGAs where custom programmed to perform preliminary computations in order to reduce the data leaving a disk to the computer. The conclusions of the study were that both Hadoop and Active Drive systems scaled well for increasing data sizes. The disadvantage of Hadoop clusters was that they required a dedicated system for the distributed file system which could not easily be used for any other task involved in a datacenter.

LVFS on-drive computation method is similar to this study. The main differences are that their methodology required custom programmed FPGAs to perform computation whereas LVFS is capable of performing general purpose computation. LVFS also utilized a commonly used distributed computing programming model, MapReduce.

3.14 Dynamic Non-Hierarchical File Systems for Exascale Storage

In this report, the authors propose to abandon the POSIX directory hierarchy in favor of a Non-Hierarchical file system (Long & Miller 2015). The authors state that because the POSIX method of managing and naming files was developed 40 years ago, many modern systems have to augment the file system design with external layers, such as databases or ”semantic extensions on top of existing file systems”.

The authors propose to develop a abandon current file system structures and develop a radically different structure which incorporates provenance and rich semantic metadata in addition to storing file content. The proposed file system would primarily rely on searching methods to locate files rather than traditional hierarchical directory tree. The authors argue that this way users would be able to find the data of interest in a more efficient way by searching for both content of files as well as the history of the files. The authors propose to completely eliminate the hierarchical tree structure of a file system and replace it with a search interface. To achieve this they argue they need both accurate indexing as well as scalable indexing. Additionally, the authors propose to develop a metadata clustering concept to manage the metadata on a disk. They propose to store metadata information close to the data to leverage faster sequential read performance from disks instead of the random access currently required to access metadata for a file.

Chapter 4

THE LIGHTWEIGHT VIRTUAL FILE SYSTEM APPROACH

At it's core the Lightweight Virtual File System (LVFS) consists of a combination of different plugins which, when chained together, provide the POSIX file system interface. The LVFS core code is responsible for loading the desired plugins and directing POSIX requests to the appropriate chain of plugins for execution. To achieve quick development and greater flexibility than most other file systems, LVFS is developed on top of Filesystem in Userspace (FUSE) (libfuse 2014).

Figure 4.1 shows the traditional Linux Virtual File System (VFS) software stack. With the standard VFS software stack file systems are developed as kernel modules and loaded into the kernel at boot time. Communication between user applications and the file system is accomplished by having an application perform POSIX file system calls via the GNU C-Library (glibc) which passes the requests to the kernel and then to the file system kernel module via the VFS. Figure 4.2 shows the FUSE software stack. FUSE implements a kernel module similar to other file systems. However, unlike other file systems, the FUSE kernel module redirects requests from the kernel back to a user program. This program is responsible for responding to the original file system requests. With standard linux file system modules a user application communicates with a kernel module. With FUSE a user application communicates with another user application.

The usage of FUSE in LVFS adds another layer of indirection which results in additional overhead. The design decision to use LVFS included the acceptance of this performance penalty. The goal of LVFS is to have fast aggregate bandwidth. Performance results shown in later sections show that this performance result are not significant for read operations. The benefits of having a flexible and easily upgraded file system as a result of

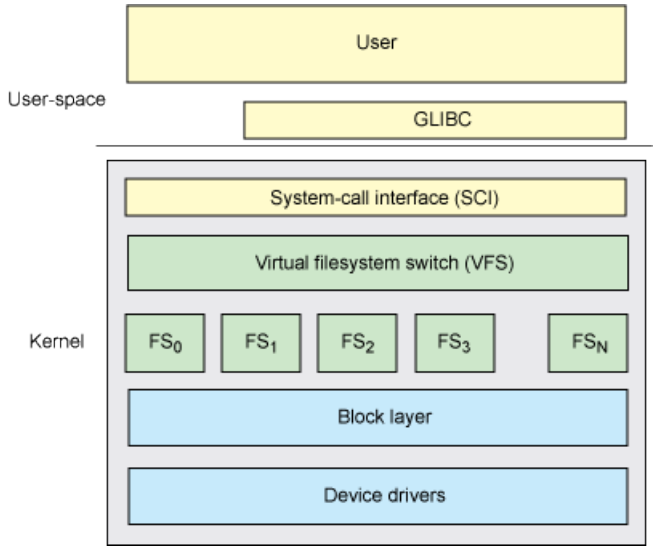


Figure 4.1: The Linux Virtual File System (VFS) layer used to access different underlying storage systems (Jones 2009)

FUSE outweigh the performance penalties incurred by its usage. However, the design of LVFS is not reliant on FUSE. LVFS can be modified to be a standard file system module inside the linux kernel.

The following sections we discuss various aspects of LVFS. In section 4.1 we give a brief overview of the LVFS architecture. Next in section 4.2 we discuss the various plugin categories along with existing plugins. Finally, in section 4.3 we discuss how LVFS configuration is used to achieve a flexible file system with all the features described in section 2.2.

4.1 LVFS Overview

As explained in section 4, LVFS is currently implemented as user space application utilizing FUSE to communicate with the Linux Kernel's VFS. Figure 4.3 shows the software stack in relation to FUSE and the different categories of plugins that LVFS manages. POSIX file system calls are transferred from a user space program to the VFS and via FUSE to the LVFS user space program. LVFS receives these POSIX calls and depending on the action identified in the calls passes the requests to one or more plugins as configured in the LVFS configuration file. The responses from the LVFS plugins are returned back to FUSE

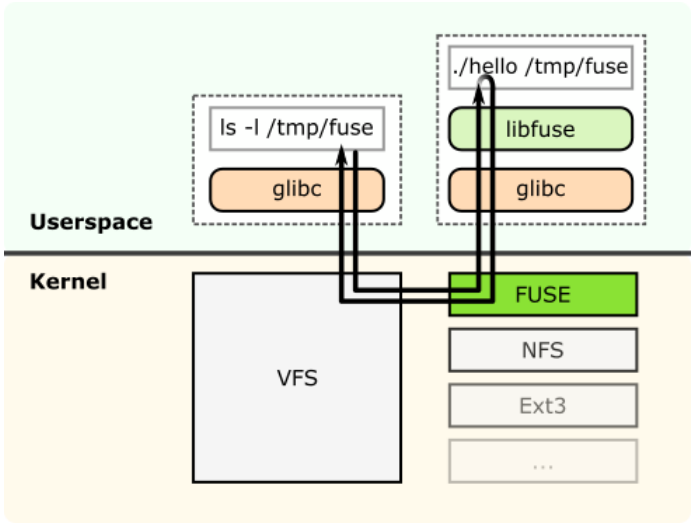


Figure 4.2: The Filesystem in Userspace (FUSE) software stack used by LVFS (Wikipedia 2017)

which returns them to the original calling program. The plugins, which we will discuss in detail in section 4.2, fall into six categories:

- Logging
- Database
- Cache
- Content
- Background
- Directory

In addition to the plugin architecture of LVFS which allows functionality to be expanded, LVFS also has a few internal capabilities available to all plugins. These capabilities include the ability to evaluate strings dynamically using the Lua programming language (Ierusalimschy, De Figueiredo, & Celes Filho 1996), the ability to evaluate conditional true/false statements. We will discuss these features along with the configuration capabilities of LVFS in section 4.3.

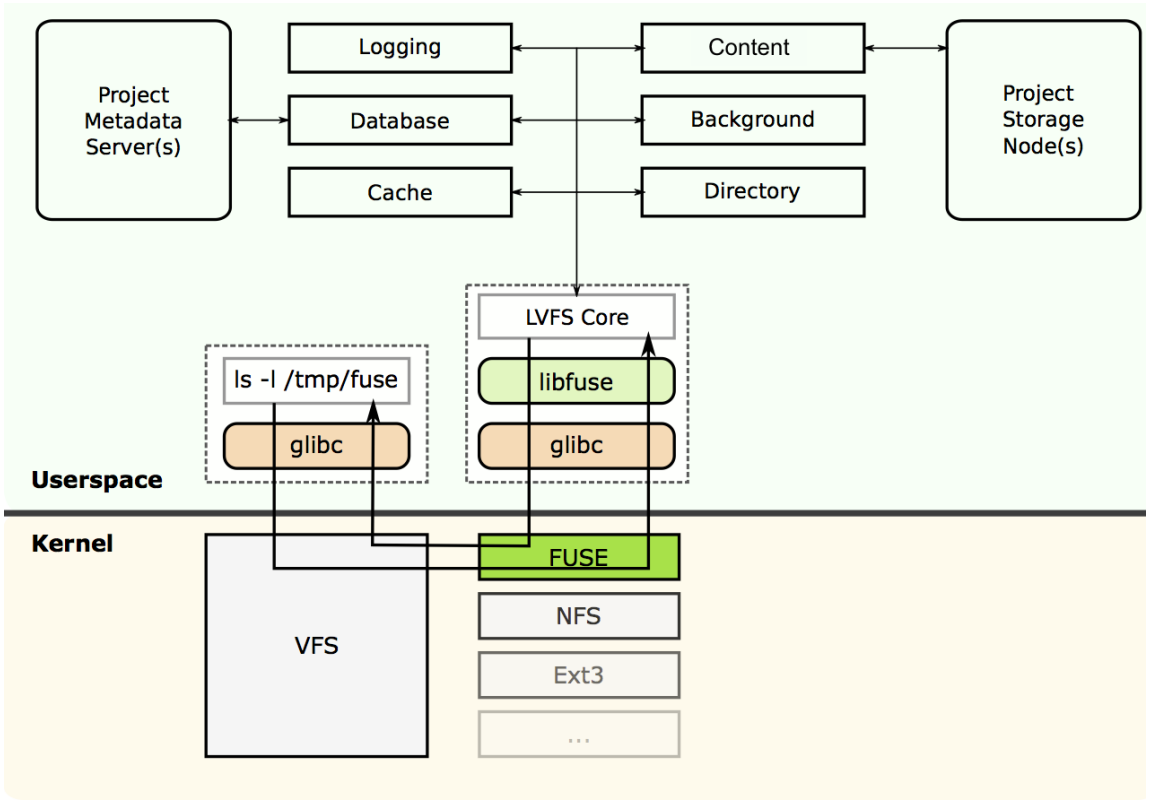


Figure 4.3: The LVFS software stack and plugin categories

To improve performance LVFS is implemented to be both multi-threaded and event based. By default FUSE uses multiple threads to process requests. Each POSIX request is handled by a different thread, up to a maximum number of threads. FUSE can be configured to run single threaded but because performance suffers, multi-threading is enabled by default. In addition to the FUSE threads, LVFS handles many threads internally to improve performance further. Generally these threads perform tasks which are usually slow, such as communicating with a metadata server. LVFS uses a thread pool worker model for threading. The pool of workers is dynamically grown and shrunk based on demand with configurable minimums and maximums.

Certain aspects of LVFS use event based programming model to improve efficiency (Dabek *et al.* 2002). An example of when LVFS uses event based programming is when retrieving data from a remote location using the HTTP protocol. Using event based programming LVFS is capable to handle multiple downloads in a single thread and use connection pooling when possible to reduce the number of network sockets it uses as well as avoid the amount of time it would take to establish a new network connection.

Figure 4.4 shows a more detailed view of the LVFS stack. The core LVFS code interprets POSIX requests and processes them based on the requests. If the results necessary to answer the request are available in short term or long term caches, LVFS will use those results. Otherwise the requests are sent to the appropriate plugin which populate the cache. The results are then sent from the cache to the FUSE library. Logging modules are used by all other modules for logging information. Similarly, background modules have access to all other modules to retrieve any desired information.

4.2 LVFS Plugins

As mentioned in section 4.1, LVFS has 6 categories plugins can belong to. In this section we will discuss the purpose of each of the plugin categories and give examples of plugins which are already implemented. Since plugins are dynamically loaded at startup based on information from the LVFS configuration, the goal of LVFS is to encourage third parties to develop their own plugins which address their specific needs or the needs of the community in general.

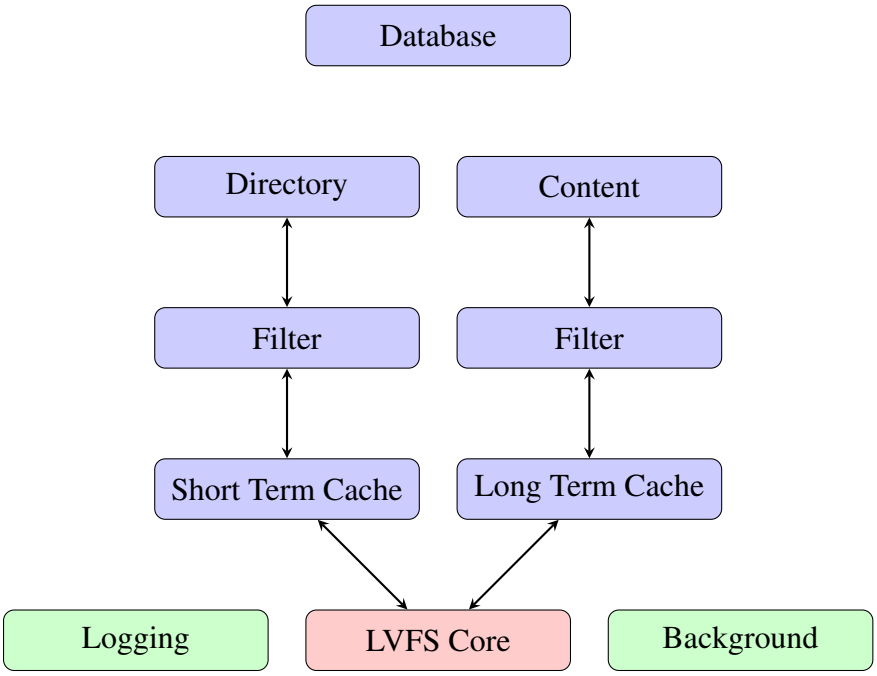


Figure 4.4: LVFS Software Stack

4.2.1 Logging

As the name suggests, logging plugins are responsible for writing log information. Most other file systems are very quiet regarding their internal functionality. LVFS on the other hand can be extremely verbose. Both the LVFS core code and any LVFS plugins with the exception of logging plugins can use the log API to report messages to be logged. Different plugins determine where the information is logged. By default LVFS has been developed with three types of logging plugins. The stdout plugin logs information to the standard output of the terminal. The file plugin logs the output to a file. Finally, the syslog plugin sends the log stream to the system logger.

Log modules can be configured with three different log levels ranging from debug statements to critical errors. Additionally, another unique aspect to LVFS is the flexibility of logging. LVFS is capable of logging to multiple sources at different log levels.

4.2.2 Database

Database plugins allow LVFS to establish connection with different metadata sources. By default LVFS comes with plugins to support PostgreSQL, SQLite (Owens & Allen 2010), and CURL. Plugins in this category are used as data sources for other plugins. They abstract the details of the communication between metadata sources and other LVFS plugins. These plugins ensure LVFS can use any future metadata sources such as NoSQL (Leavitt 2010) databases like redis (Labs 2017), MongoDB (Inc. 2017), etc. (Han *et al.* 2011). Plugins such as those responsible for generating directory layouts or those responsible for determining file content can use these plugins to retrieve the necessary information for their output.

4.2.3 Cache

Cache plugins are a very important part of LVFS. POSIX file systems assume generating directory structures are a fast task. However, since LVFS does not maintain its own internal metadata structure, it can't guarantee the speed at which metadata is retrieved. To ensure a fast response time, LVFS utilizes cache modules to temporarily store results retrieved from other sources. Cache modules are the most complex plugins in LVFS since they need to execute as fast as possible. LVFS supports two types of cache categories and all plugins can be assigned to either category as desired by the data center or the project. Long term cache is used to cache the contents of files for quicker local access. Short term cache is used by LVFS to cache retrieved metadata information. Typically files will be significantly larger than metadata information so most of the time a disk based cache plugin will be used for long term cache while a memory based cache will be used for short term to improve performance.

4.2.4 Directory

Directory plugins are responsible for generating file and directory listings. Directory plugins utilize database plugins to retrieve information for this task. These plugins utilize an API that is a simplified version of the POSIX file system calls. LVFS comes with plugins for generating file and directory structures based on SQL queries, HTTP or FTP listings, Object based APIs such as S3, and any future plugins developed for such tasks.

4.2.5 Content

Content plugins retrieve data from storage nodes when files are accessed for reading or writing. By using a plugin system for file content, LVFS is capable of storing and retrieving files from a wide variety of storage systems and future systems. LVFS comes with plugins capable of retrieving file content from block based disks, object based drives such as Amazon S3, DDN WOS, or HGST AAS. LVFS also has plugins for generating file content from more unusual sources such as from a database.

4.2.6 Background

Background plugins are generic plugins that fit in none of the other categories. LVFS does not directly interact with these plugins beyond loading them and initializing them to execute in the background. These plugins can perform any task that is not directly related to handling POSIX requests. An example plugin is the GMetric plugin which collects statistics information from all other LVFS plugins and the core LVFS code and sends the collected statistics to a Ganglia system for graphing. This allows LVFS to be monitored by an operator.

4.2.7 Filter

Filter plugins are used to for features such as streaming format conversions. These plugins intercept the output produced from Directory, Content, or other filter plugins and alter their responses. An example plugin is a plugin capable of converting HDF (Folk *et al.* 2011) to GeoTIFF (Ritter *et al.* 2000). The plugin intercepts the response from the Directory module which created the file listing containing the original HDF files adding the GeoTIFF files to the listing. When GeoTIFF files are opened for reading the module intercepts the read calls and performs a read of the original HDF file and converts the data GeoTIFF passing the GeoTIFF data to the user.

4.3 LVFS Configuration

The flexibility of LVFS not only stems from the use of the above described plugins but also from the configuration file which directs LVFS on how to chain the various plugins together to create the desired directory layout. In most linux file systems most of the

information provided here is stored in inodes and superblocks (Jones 2009). Because LVFS stores the information in a human readable format it gives the data center and the project a great amount of flexibility on how they structure files and directories and how they retrieve the files for those files. LVFS configuration is stored in YAML format (Ben-Kiki, Evans, & Ingerson 2005).

On startup, LVFS loads the YAML configuration file. This configuration file lists all plugins which should be loaded into memory and how each module should be configured. The LVFS configuration file, as shown in listing 4.1, is divided into two sections. The first section, called the *Modules* section, is the LVFS equivalent of a standard block based file system's superblock. The second section, called the *root* section, has similarities to the inode structure.

The *Modules* section tells LVFS, as the name describes, what modules to load, what to call them, and what global options should be passed to the modules. For example, line 2 in listing 4.1 tells LVFS to load a plugin named *Syslog* and assign an alias name of *default*. The plugin is loaded with the log level of *info*. In the *root* section, line 17 tells LVFS where to mount the file system. Lines 18 and 19 tell LVFS which modules to use for long term and short term caching. The names used are the names of plugins loaded on line 6 and 7. Because LVFS configurations can get quite large, LVFS utilizes the YAML tag feature to add the ability split the configuration file into multiple files. Line 20 shows this ability by including the remainder of the LVFS configuration in a separate file, *root.yaml*.

Listing 4.1 only shows how modules are loaded. Listing 4.2 shows a sample configuration which creates a static file and a directory. The file as described on lines 1–6 is named *README* and the content is the string provided on lines 5 and 6. Lines 7–9 define a directory named *myDir*. Inside this directory, when a listing is requested via a POSIX call, LVFS will query the database using the query provided on line 12. We will discuss the details of this and other queries in chapter 5.

```
modules:
  - { name: Syslog, alias: default, level: info }
  - { name: CURLConnectionPool,  alias: curlPool }
  - { name: PostgresDB, connectString: service = laads-lvfs }
  - { name: SQLiteDB }
  - { name: MemorySwap, cacheTime: 600 }
  - { name: DiskSwap, path: /MODcache }
  - { name: S3Access, accessKey: modis, accessID: modis }
  - { name: SQLContent }
  - { name: GMetric, reportInterval: 5 }
  - { name: Dir }
  - { name: File }
  - { name: SQLFile, database: PostgresDB }
  - { name: SQLDir, database: PostgresDB }
  - { name: JPGFilter }
root:
  mountPoint: /mnt
  longTerm: DiskSwap
  shortTerm: MemorySwap
  children: !lvfs:include root.yaml
```

Listing 4.1: Sample LVFS Configuration file

```
- type: File
  name: README
  locator:
      string: >
        Congratulations! Your LVFS setup works.
        To configure LVFS please take a look at /etc/lvfs/lvfs.conf
- type: Dir
  name: myDir
  children:
    - type: SQLFile
    query:
      statement: select name, size, time from file where dir = $1
      bindings: [ "${-1}" ]
```

Listing 4.2: LVFS Configuration defining a simple directory and file

Chapter 5

THE LIGHTWEIGHT VIRTUAL FILE SYSTEM IMPLEMENTATION

The LVFS implementation is mainly C++ with some C code where necessary and some python utilities for maintenance. The code is split into two components; (1) the core LVFS code and (2) the plugin API. The core LVFS code interprets the POSIX requests and calls various plugins to collect the necessary information as a response to the original POSIX request. The core code is written using C++ object-oriented approach and uses C++11 standards including threading, regular expressions, exception handling, move semantics, and data structures. The plugin API is provided as a mostly C++ class API. Some functionality lacking in C++ is implemented as C calls. The LVFS core code provides base classes for all the different categories of plugins. LVFS plugins extend these base classes and implement the necessary functions to perform their tasks.

5.1 LVFS Core

The LVFS core implements the low level callbacks used by the FUSE library to execute POSIX file system requests. LVFS core code is mainly responsible for parsing these requests and calling necessary plugin functions for the information necessary to honor the requests. Additionally, the core LVFS code provides some commonly used functionality as both a convenience to plugin developers as well as a common communication API between plugins and LVFS. The components in the LVFS core are Plugin Management, Access Control Lists (ACL), Backtracing, Content Management, File/Directory Management, String Management, Condition Evaluation, and Configuration Parsing.

Category	Type
Log	Global
Background	Global
Cache	Global
Content	Global
Database	Global
Directory	Instance
Filter	Instance

Table 5.1: List of LVFS plugin categories and their instance types

5.1.1 Plugin Management

One of the core aspects of LVFS is the management of plugins. LVFS utilizes two types of plugins; *global* and *instance*. As the names suggest, *global* plugins are plugins which exist globally. LVFS will only initialize a single global instance of these plugin types. *Instance* plugins, on the other hand, do not exist globally but have multiple instances initialized based on the LVFS configuration. The plugin type is determined by the plugin category. Table 5.1 shows a list of plugin categories and their instance types.

Global plugins are configured once globally and used throughout the LVFS code. The core LVFS code or plugins can request access to any global plugin. Global plugins are configured during the *module* section as described in section 4.3. *Instance* plugins are loaded into memory by the module manager at startup similar to *global* plugins. However, unlike *global* plugins, these plugins are not initialized until the *root* section. In the *root* section, instance plugins are provided with additional options which can be additions to or replacements of global options. A new instance of the plugin is initialized each time a plugin is mentioned in the *root* section.

When the plugin manager parses the *module* section, it will use the *name* attribute to locate the plugin file in the directory configured into LVFS at compile time. Additionally, a plugin can be referenced in other sections by using the same *name* attribute. The plugin manager allows for loading of the same plugin multiple times with different options. The *alias* attribute can be used to differentiate between plugins with the same *name*. Listing 5.1 shows an example where the same plugin is loaded twice. The plugin manager will locate the *PostgresDB.so* file in the LVFS directory specified at compile time. Each time

```
1  modules:
2    - { name: PostgresDB, alias: db1, host: projDB1 }
3    - { name: PostgresDB, alias: db2, host: projDB2 }
4
5    - { name: SQLFile, database: db1 }
6  root:
7    children:
8      - type: SQLFile
9        query: select name, size, time from file
10     - type: SQLFile
11       database: db2
12       query: select name, size, time from file
```

Listing 5.1: LVFS configuration loading same plugin twice

the plugin is loaded the remaining attributes are provided as arguments to the plugin. In this case the plugin is loaded twice. The 1st instance is configured to connect to the host *projDB1* and the 2nd instance is configured with host *projDB2*. In order to differentiate the two instances, the *alias* attribute provides an different name for each. In the *root* section we can see an example of how the two instances could be used. We have initialized two modules to populate the top level directory with files retrieved from two different databases which happen to have the same schema.

Continuing with the example in listing 5.1, the plugin manager will load a Directory plugin called *SQLFile*. Table 5.1 tells us that Directory plugins are *instance* plugins. This means that at startup the plugin manager will load the plugin into memory and remember the global options specified. However, the plugin manager will not initialize the plugin. When parsing the *root* section. The Directory Manager, described in more detail in section 5.1.4, will request an instance of this plugin to be initialized. In this case, the directory manager will request an instance of the SQLFile plugin to be instantiated. In the 1st instance, the global configuration is used and an SQL query is added to the configuration. For the 2nd instance, the same query is also added to the configuration. The *database* attribute, however, is overridden from the global one. This way the same plugin can be used to connect to two different databases for creating a single directory listing.

```
1 - type: File
2   name: myFile
3   acls:
4     - user::rx
5     - group::r
6     - user:user1:rx
7     - group:group1:rx
8     - user:user2:rwx
```

Listing 5.2: Sample LVFS ACL

5.1.2 Access Control Lists

LVFS Access Control Lists (ACL) are defined similar to the way they are defined by the POSIX 1e. acl specs (The Open Group 2017; Grünbacher 2003). The Access Control List class in LVFS is a combination of the standard unix permission system and the ACL system supported by many other file systems. The Access Control List class is part of the base class providing the directory plugin APIs. With this implementation all plugin implementation automatically have the ability to support permission based access to files and directories.

The ACL component can be triggered for any directory module by including the *acls* attribute. The value of the attribute is a list of ACL definitions. If no ACLs are defined the default values are to set ownership to the administrator user and give read only permissions to everyone. Additional permissions can be specified using the same format as Linux ACLs.

Listing 5.2 shows some possible ACL values. Lines 3–8 define the ACL values for the defined file instance. Lines 4 and 5 override the default permissions of the file specifying that the default user, which is the administrator account, will have read and execute permissions and the default group will have read permissions. Additionally, the user *user1* and the group *group1* are given read and execution permission on the file. Finally, user *user2* is given read, write, and execute permissions.

5.1.3 Backtracing

The LVFS core code contains backtracing capabilities. On startup the LVFS code registers signal handlers which are triggered during critical events, such as segmentation faults. When such an event is triggered, the backtrace code performs a stack trace of the

current state of LVFS and writes the state out as critical events on the registered log module. This provides developers valuable information for debugging live systems to improve reliability of LVFS and its plugins.

5.1.4 File/Directory Management

The directory management code in LVFS is responsible for generating file and directory listings from the provided configuration file. This is done by providing a combination directory plugins, described in the section 5.2.4, in a tree like structure which simulate the directory tree listing. A single node in the directory tree consists of the mandatory option *type* and the optional option *children*. The *type* option specifies which directory plugin should be used for the current location. Subdirectories are specified recursively by using the *children* option. The *root* section as described in section 4.3 always contains the *children* option under which file and directory plugins are listed.

Additional options are specified depending on the plugin used at any given directory level. These options are defined by each plugin. Failure to provide required options for a particular plugin results in error conditions at startup. Options, which are not used by the plugin, are silently ignored.

The directory manager will receive POSIX requests and the paths for the requests. We call these the *input paths*. Input paths are provided relative to the mount point under which LVFS is mounted as explained in 4.1. For example, assuming LVFS is mounted under */mnt*, and a POSIX request for a directory listing of */mnt/some/path/to/list* is sent to LVFS then the *input path* is */some/path/to/list* and the POSIX request is a directory `listing`. Similarly, if a request comes in to open */mnt/some/path/to/a/file* then the POSIX request to the directory manager is `open` and the *input path* is */some/path/to/a/file*.

Listing 5.3 shows a sample configuration which continues the example provided in listing 4.1. The *root* section on line 1 has to contain the *children* keyword, line 5, to indicate where the file/directory layout begins. Lines 6–9, define the layout. The layout consists of two plugins at the same level, starting at level 1. The first plugin is of type *Dir* and the second one is of type *File*. The remaining attributes in each section are specific to each plugin. In this case both the File and Dir plugin require an attribute called *name*. When a directory listing is requested of the directory /mnt, the directory manager will receive a `listing` POSIX request with input path of /. Each / in the input path determines

```
1 root:
2   mountPoint: /mnt
3   longTerm: DiskSwap
4   shortTerm: MemorySwap
5   children:
6     - type: File
7       name: myFile
8     - type: Dir
9       name: myDir
```

Listing 5.3: Simple File listing

the number of levels in the input path. The directory manager will traverse the directory layout tree up to the level of the input path. In this example the input path is level 1 and the layout starts at level 1 so the directory manager will immediately stop traversal. All plugins available at the level at which traversal is stopped are queried for input for listings. The results of the queries are concatenated and are used as the final answer for the original POSIX request. In this case the directory manager will stop at the top level and query the *File* and *Dir* plugins for listing resulting in a directory listing with a file named *myFile* and a directory named *myDir*

Algorithm 1 shows the pseudo code used by the directory module to traverse the layout tree for matching an input path. At each level of the layout tree, the directory manager queries the plugins at that level in the order they were defined in the configuration file. The first plugin to accept the *path element* from the *input path* at that plugin's level is used as the owner of that *path element*. The directory manager recursively continues to traverse the tree until the input path is exhausted or no plugin is found to be responsible for a given path.

Listing 5.4 shows a three level layout configuration example. If a POSIX directory listing request is received with an input path of */myDir/myDir2* then the directory manager will query the plugins at the first level. The directory manager will query each plugin at the same level in the order provided in the configuration. The first defined plugin at level 1 is the File plugin When queried for ownership of *myDir* it will return false. The second plugin is the Dir plugin which will return true for the same query. Therefore, the directory manager will recursively traverse down the children of the 2nd plugin. It will query the 1st child of the 2nd plugin for ownership of *myDir2* which will return true as well. Since the

Algorithm 1 Directory Layout traversal to find plugin responsible for input path

```
1:  function TRAVERSE-LAYOUT(currentLevel, currentPlugin, inputPath)
2:      if EVALUATECONDITION then
3:          if inputPath[currentLevel] ∈ currentPlugin.listing then
4:              if currentLevel = inputPath.levels then
5:                  return currentPlugin
6:              else
7:                  for child ∈ currentPlugins.children do
8:                      return TRAVERSE-LAYOUT(curentLevel + 1, child, inputPath)
9:                  end for
10:             end if
11:         end if
12:     end if
13:     return Not Found
14: end function
```

directory manager has exhausted all paths in the input path it will terminate at this point and return the 1st child of the 2nd plugin as the result of the traversal.

The directory manager performs several other tasks as well such as ACL and condition evaluations. We will briefly discuss these tasks and how the directory management utilizes them. Details of these components is provided in sections 5.1.2, and 5.1.7.

ACL settings are evaluated partially by LVFS and partially by the FUSE library. Since FUSE evaluates each input path individually, the directory manager only needs to evaluate the last element of the input path for ACL permissions. Therefore, the directory manager will evaluate the ACL permissions after the layer traversal has finished and found the correct plugin.

Condition evaluations are different from ACL as they are not a POSIX concept. As shown in algorithm 1 conditions are evaluated by the directory manager first. If a condition string is evaluated to false the directory manager aborts traversal and no further children are traversed.

The core directory manager interprets POSIX attribute requests for input paths and, after locating the proper plugin, queries the plugin for the POSIX `stat` information. Standard POSIX structure includes information such as the size of the file, the permissions of the file, the type of file, etc. Most of these are usually not filled out by the directory plugins and LVFS provides reasonable default values for them. The two important fields that are

```
1  root:
2    mountPoint: /mnt
3    longTerm: DiskSwap
4    shortTerm: MemorySwap
5    children:
6      - type: File
7        name: myFile
8      - type: Dir
9        name: myDir
10       children:
11       - type: Dir
12         name: myDir2
13         - type: File
14           name: myFile2
```

Listing 5.4: Simple File listing

```
1  - type: File
2    name: myFile
3    statByContent: true
4    locator:
5      deferred: false
6      location: https://google.com/index.html
```

Listing 5.5: Example telling directory manager to query locator for stat information

filled out by plugins are the size of the file and the type. The type determines if an object is a file or directory. In the case of a directory, the size is ignored so it is usually not set by those plugins. Under some circumstances the directory plugins are not able to provide all POSIX `stat` information. In those cases, a flag in the LVFS configuration specifies that the directory manager should query both the directory plugin as well as its locator information, described in more detail in section 5.1.5, for `stat` information. The directory manager will first query the directory plugins and then let the content manager, described in section 5.1.5, override the information set by the directory plugin. Listing 5.5 shows a sample LVFS configuration notifying the directory manager to query both the directory plugin as well as the content manager for `stat` information. Line 3 is the setting which enables this behavior. The lines following line 3 define the location information for the content manager to use to retrieve the file content. We will discuss this in detail in section 5.1.5.

5.1.5 Content Management

Section 5.1.4 describes how to create file and directory listings but it does not mention how LVFS obtains or generates the contents of files. The content management code in LVFS is responsible for the task of building the content of any file opened for access. LVFS uses the concept of *Locators* and *Locations* to evaluate the content of a file. A *Locator* is a set of one or more *Locations* which combined determine the content of a file. A *Location* is the definition of a data storage. This definition can be the location of a file on a local disk, a remote disk, the result of an SQL query, or any other resource represented in URI style (Berners-Lee, Fielding, & Masinter 1998; Berners-Lee, Fielding, & Masinter 2005).

Locators use multiple locations for two purposes; (1) backup locations of the same data or (2) concatenation of different data into a single view. *Locators* can also provide a level of indirection by allowing a location to point to another location where the desired content is. These are called *deferred* locations. An example would be an SQL query used as a *deferred* location whose result represents an URL where the file content is actually located.

Plugin developers write simple content module plugins, defined below, which are capable of retrieving content from one or more *locations*. For example, an **HTTP** content plugin would perform **HTTP** downloads of data while an **S3** content plugin would implement the ability to remotely download from an **S3** based object storage device. The content manager queries each content plugin for the *schemes* supported by that plugin. The content manager, knowing the *schemes* supported by each plugin, can pass locations to the proper module for retrieval.

All of this information is encoded in the LVFS configuration file in YAML. Each *locator* is provided with the necessary attributes to let the content manager know how to resolve the *locations*. These attributes are *deferred*, *any*, *concat*, and *location*.

The *deferred* attribute is a boolean with a default value of *false*. If set to *true*, it informs the content manager that the following *location* will contain the location of the file content *after* evaluating the original *location*. If left as *false*, the *location* represents the location of the content of the file.

The *any*, *concat*, and *location* are mutually exclusive attributes. The *any* attribute contains a list of *locations* which tell the content manager that any of the locations can be used for content retrieval. LVFS' content manager will randomly chose a *location* from the given

list and retrieve the content from there. Should the chosen *location* be unavailable, LVFS will continue to randomly pick another *location* until all *locations* have been exhausted or the content is retrieved successfully. The *concat* option tells LVFS that the content from the list of *locations* should be combined to form the final content of the file.

Listing 5.6 shows several examples of *locators* and *locations*. Lines 4–6 define a *locator* with a single *location* which is not *deferred*. The *location* will use the content plugin which supports the **HTTPS** scheme. Therefore, on file access, LVFS will retrieve the `index.html` page from `google.com` and provide it as content for the file. Lines 10–15 define another non-*deferred locator*. This time, the *any* attribute tells LVFS that the two provided *locations* will serve as alternate locations. In this example, LVFS is instructed to randomly pick between the content of the two web pages and provide the first successful one as the content of the file. Finally, the example on lines 20–25 instructs LVFS to concatenate the content of three *locations* and provide that as the content of the file. In this example a plugin responsible for **HTTPS**, one responsible for **FTP**, and one responsible for **S3** downloads all provide the final content.

In most situations all the flexibility of the content manager is not needed. In order to reduce verbosity of the LVFS configuration there are shorthand versions that can be used. Listing 5.7 shows the shorthand versions of the lines 4–6 and 10–15 from listing 5.6.

The examples of *locators* so far assume that the location of the files are known at configuration time. Frequently, however, the location for the content of files need to be determined at run time. An example of such a scenario is when the location of a file is stored in a database. For those cases, the content manager provides *deferred* locations. *Deferred* locations are locations to locations. Listing 5.8 shows an example of such a *locator*. In this example LVFS' content manager will first contact the database plugin named as postgresql01, perform the given query. The result of that query is then used as the *location* which the content manager will utilize similar to the above examples.

So far we have only discussed reading file content from a remote source. Storing content back to a remote source follows the same logic with limited support. Any *locators* specified which use *concat* are not allowed to have write support. *Locators* using *any* will instruct LVFS to only store the file content to the first *location*. LVFS makes the assumption that the remote storage node is responsible to distribute the file to the other alternate *locations*. Similar to *concat locators*, *deferred locators* also do not have write support with current LVFS implementations. The reason for this design decision is that

```
# Single location
- type: File
  name: myFile
  locator:
    deferred: false
    location: https://google.com/index.html

# Content can be retrieved at one of these locations
- type: File
  name: myFile
  locator:
    deferred: false
    any:
      - "http://www.rfc-editor.org/"
      - "https://www.rfc-editor.org/"

# Content is the concatenation of multiple sources
- type: File
  name: myFile
  locator:
    deferred: false
    concat:
      - https://google.com/index.html
      - ftp://ftp.debian.org/README
      - s3://cluster00/BUCKET/README
```

Listing 5.6: Sample LVFS Locators and Locations

```
- type: File
  name: myFile
  locator: "https://google.com/index.html"

- type: File
  name: myFile
  locator:
    - "http://www.rfc-editor.org/"
    - "https://www.rfc-editor.org/"
```

Listing 5.7: Shorthand Locators

```
1 - type: File
2   name: myFileName
3   locator:
4     deferred: true
5     location:
6       db: postgresql01
7       query: select location from files where name='myFileName'
```

Listing 5.8: Deferred Locators

both *concat* and *deferred* locators are generally used with database information. Projects usually have robust pipelines to ensure their metadata databases have proper information and generally would dislike a system not directly under their control, like LVFS, to write into their metadata database.

Beyond those limitations to write support plugins also can either be read only plugins or have write support. Content plugins are described in more detail in section 5.2.7.

5.1.6 String Management

In section 5.1.5 we described *deferred* locations as a way to dynamically define locations at runtime. While that option can be used for some circumstances there are other circumstances where other methods are necessary. For those scenarios, LVFS offers a feature rich string class. LVFS supports keyword replacements which are performed dynamically at runtime. These keywords can be used to modify the strings prior to evaluation. LVFS string sections are marked as needing evaluation by encapsulating the string section in *${}*. LVFS strings are evaluated using the Lua language (Ierusalimschy, De Figueiredo, & Celes Filho 1996) with some extensions to reduce the verbosity of the strings and reduce complexity.

Dynamic string manipulation is the primary method by which LVFS is capable of providing dynamic directory layouts. LVFS strings replaces patterns such as *${1}* or *${2}* with the corresponding levels from the input paths. This way LVFS strings can dynamically change based on paths being queried. Listing 5.9 shows an example where the directory listing dynamically changes based on the input path. Table 5.1 is the sample data used for the LVFS configuration. A request for a directory listing which results in input path of `/` would return three directories consisting of `Washington`, `Baltimore`, and

```
1 - type: SQLDir
2   query: select Name from locations
3   children:
4   - type: SQLFile
5     query: select name, size, time from file where location="${1}"
```

Listing 5.9: LVFS Configuration showing dynamic string modification

Name
Washington
Baltimore
Annapolis

(a) Locations Table

Name	Size	Time	Location
File1	10	1234	Washington
File2	10	1234	Baltimore
File3	10	1234	Annapolis

(b) File Table

Figure 5.1: Sample data to used with the dynamic query in Listing 5.9

`Annapolis`. This is generated by the first plugin instance. When a user requests a listing of any of the directories, say `Washington`, then the first child of the first plugin instance will be invoked. Prior to passing the string to the database for querying, LVFS will perform dynamic string transformations. In this case, LVFS will detect the *${1}* string and replace it with the first path element of the input path. In our example, the input path is */Washington*, therefore, *${1}* is replaced with the string `Washington`. The resulting query is *select name, size, time from file where location="Washington"*. The listing of */Washington* will be *File1*. Similarly a listing of */Baltimore* results in the query *select name, size, time from file where location="Baltimore"* and the directory listing of *File2*.

LVFS dynamic strings are not limited to positive integers. Negative values and a value of zero are supported as well. LVFS treats the input path as an array of path elements with each element separated by / and array values starting at 1. LVFS uses the value 0 as a special value which always represents the path element assigned to the current plugin. Using the LVFS configuration from listing 5.9 and an input path of */Washington/File1*. Had the value of *${0}* been used in the query on line 2 then LVFS would have replaced it with the string `Washington`. At the same time if the path further evaluates to the next plugin on line 5 then the same *${0}* would evaluate to *File1*. Negative numbers evaluate backwards starting one path element above from *${0}*. Using the same input path as before if line 5 contained the value *${-1}* then it would have evaluated to the value of `Washington`.

```
1  - type: File
2    name: myFile
3    locator:
4      - http://192.168.0.100/myFile
5      - http://192.168.0.101/myFile
6      - http://192.168.0.102/myFile
7      - http://192.168.0.103/myFile
8  - type: File
9    name: myFile2
10   locator: http://192.168.0.${tostring(math.random(100,103))}/myFile
11 - type: File
12   name: myFile3
13   locator: http://192.168.0.${random(100,103)}/myFile
```

Listing 5.10: LVFS Configuration showing Lua function use

In addition to allowing placeholders for input path replacement, LVFS also supports execution of Lua code. To reduce the chances of an exploit being used as an attack vector into the operating system, LVFS limits Lua access to math and string libraries in Lua. Other functions, like operating system access functions are disabled. To simplify configuration, some commonly used operation have LVFS defined versions. Listing 5.10 shows how Lua functions can be used to simplify configuration. Lines 1–7 show a file whose contents exist on four different servers accessed. LVFS will randomly pick between the four alternate locations of the file and download it. Lines 8–10 similarly pick from the same four hosts randomly but using Lua in a less verbose manner. Lines 11–13 use Lua with some LVFS provided convenience functions which combine the *tostring(math.random())* function sequence into a single Lua function call.

5.1.7 Condition Evaluation

Condition evaluations, are briefly described in section 5.1.4. Condition evaluations have no POSIX file system equivalent. The purpose of condition evaluation is to make certain parts of the directory structure be only visible under certain circumstances. Similar to ACLs and LVFS Strings, condition evaluation is available to all directory plugins and is evaluated for all plugins automatically since it is part of the core LVFS code.

Listing 5.11 shows a common way condition variables are used. This example uses the same data as shown in table 5.1. In this example, the data center or project wishes

```
1  - type: SQLDir
2    query: select Name from locations
3    children:
4    - type: SQLFile
5      query: select name, size, time from file where location="${1}"
6    - type: File
7      name: Readme
8      condition: "${1}" == "Baltimore"
9      content:
10       - string: This warning is only for 'Baltimore' directory
```

Listing 5.11: Sample Condition Variable

to display a `README` file with some hard coded content but only for the `Baltimore` directory. Without the condition variables the LVFS configuration from lines 1–5 would be repeated twice. Once using a database queries excluding the directories for which there should be no `README` file and then another similar query for which there should be the `README` file. To reduce verbosity, condition variables allow LVFS to evaluate if it should allow file/directory visibility. Condition variables use the same LVFS string functionality as described in section 5.1.6. In this example the *${1}* input path is replaced by the LVFS string evaluation. After that replacement, LVFS performs a Lua evaluation which, if *true*, allows the layout tree rooted at that node to be visible for the remainder of the LVFS evaluations.

5.1.8 Configuration Parsing

The LVFS configuration is written in YAML and LVFS uses most YAML features such as objects, arrays, and strings. Because LVFS configurations can be large and to encourage configuration reuse, LVFS use YAML tag directive feature to add the ability to split configuration files into multiple files. Listing 5.12 shows an example of how a configuration can be split. On line 1 LVFS will read the *modules.yaml* and insert its content in place of the *!lvfs:include modules.yaml* tag. Similarly, for line 2, LVFS will read the *root.yaml* file and replace it at the include location. The inclusion of other configurations can be arbitrarily deep.

The combination of complex LVFS configuration and the ability for third party plugin developers to add their own configuration options for their plugins means that errors can

```
1 modules: !lvfs:include modules.yaml
2 root: !lvfs:include root.yaml
```

Listing 5.12: LVFS include directives

exist in the configuration. On startup, LVFS will parse and validate the configuration for valid syntax and valid schema. During startup LVFS will perform syntax validation. If the YAML configuration does not pass syntax validation, LVFS will exit with an error message to indicate where the syntax error occurred.

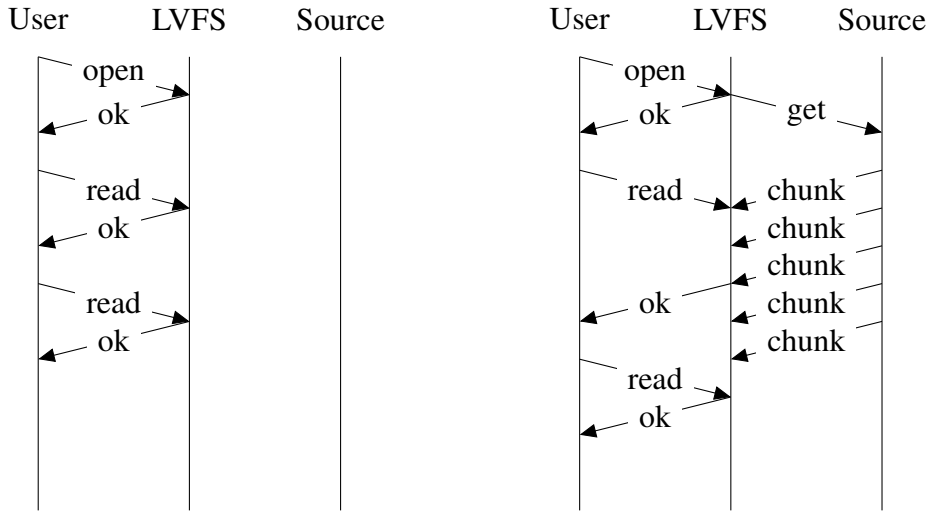
The YAML specification does not have schema validation built into the language. LVFS, therefore, implements its own schema validation. LVFS configuration options which are part of the core LVFS code are hard coded into LVFS for validation. Plugins, however, need to define their own schemas. The LVFS plugin API provides functions for plugins to use to define their schemas. For each option, LVFS plugins can define if the options are mandatory or optional, whether there are any default values for the option and a brief description about what setting the option controls. For the option values, plugins can also specify the value type. The type can be strings, integers, or complex YAML types like arrays or objects. This API is described in more detail in section 5.2

In addition to verifying syntax and schema at startup, LVFS has a python toolkit which performs the same validations independent of the executable. This allows for the verification of configuration changes prior to reloading LVFS.

5.1.9 Cache Management

A core component of LVFS is the caching system. Because LVFS relies mostly on remote information, without a robust caching system most requests would be too slow. LVFS uses caching to store both metadata information for directory listings and to store file content. LVFS uses two cache instances which are named *short term cache* and *long term cache*. *Short term cache* is used for storing metadata information while *long term cache* is for storing file content. Since metadata information will generally be significantly smaller than file content LVFS recommends a cache plugin which is backed by memory for the short term cache while recommending a disk backed cache plugin for the long term cache.

LVFS caching is divided into two components. The contents of a file or the listing



(a) File already in cache (b) File does not exist in cache

Figure 5.2: Sample scenario for opening a file

of directory is stored in cache object which we call a *cache item*. When a plugin or the LVFS core request access to an *item* they are given access via an object we call a *cache instance*. A particular object stored in cache is only represented by a single *item* but multiple *instances* can refer to an *item*. This is similar to the weak references concept used in many programming languages. With this model a cache plugin can manage its own *cache items* and has a reference count of how many *instances* are referencing an *item*. The reference counts help the cache plugin make decisions on which *items* can be timed out and removed from the cache.

A *cache item* can have several modes of operation it can be in *persistent* or *non-persistent* mode. If *persistent*, a *cache item* can also be in *reader* or *writer* mode. To understand these different modes we will show two different scenarios.

Scenario—POSIX Read: Consider a scenario where a process performs a POSIX `open` request on a file managed by LVFS. There are two possible outcomes in this scenario; (1) the file exists in *long term cache* or (2) the file does not exist in *long term cache*. Figure 5.2 shows these two outcomes.

Figure 5.2a shows the outcome where a file already exists in cache. The user requests to open the file, LVFS determines it exists in cache and creates a *cache instance* to the ex-

isting *item* in cache and associates the file descriptor with this *instance*. The *cache instance* and *item* are, in this case, put into *non-persistent* mode.

Figure 5.2b shows the outcome where a file does not exist in cache or the cache entry is stale and needs to be refreshed. From the user's point of view the two outcomes need to be identical in order to be POSIX compliant. In this outcome, upon receiving the `open` request, LVFS will return success to the user and simultaneously request the content manager to download the file content from the source. To accomplish this LVFS creates two *cache instances* to the *cache item*. One *instance* is associated with the file descriptor of the user and the other *instance* is given to the content manager. The content manager will put its *instance* into *persistent* mode. Since the cache *item* is in *persistent* mode the user's `read` requests will block until the file segments requested have been written by the content manager. This is shown in 5.2b as a delayed response to the first read request. The read results are only returned after the first three chunks are retrieved from the source. The next read request already has all needed segments in cache and, therefore, receives an immediate response.

Scenario – POSIX Writes: To understand the difference between *persistent reader* and *persistent writer* we need to consider a different scenario involving writing. Figure 5.3 shows a scenario where the user opens a file and writes some data. Again, due to a cache miss a *persistent instance* is created for downloading. Unlike last time, however, the user is performing write operations. These write operations from the user should overwrite the write operations from the content manager no matter which order they are received. This is achieved by ensuring that a write operation from a *persistent instance* can never overwrite a write operation from a *non-persistent instance*. A similar write operation is performed by the user immediately following the first write operation. In this scenario, the content manager has not finished the store operation from the first write operation.

To ensure files are always uploaded in a consistent state LVFS will store the new writes in a different place until the first upload has completed. When the content manager is performing retrieve operations from the source it will put the *cache instance* in *persistent write* mode since it will be writing to the *cache instance*. When performing store operations, the *cache instance* is put into *persistent read* mode since it will be reading from the cache to send to remote location. When a cache item has a cache instance attached to it in *persistent read* mode then all write operations from a *non-persistent instance* will trigger a

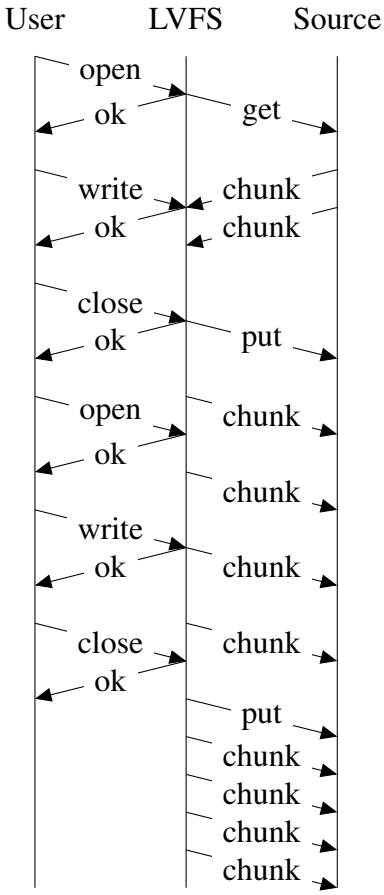


Figure 5.3: Sample scenario involving write operations

copy on write and all future reads/writes will be performed on this copy. Upon completion of the first upload, the copied data is merged back into the *cache item* and another upload is triggered. Using this method, LVFS can guarantee POSIX behavior on the local machine and guarantee uploads of consistent state only if supported by the remote end. Algorithm 2 shows how LVFS core code for cache items perform read/write operations.

5.2 LVFS Plugin Implementation

The LVFS plugin API is implemented as a series of base classes. These base classes are overloaded by plugin developers to implement their functionality. Some functions are required to be overloaded while others are optional. Some necessary functionality is not possible to be performed via C++ function overloading which requires the implementation

Algorithm 2 Algorithm for Cache Item reads and writes

```
1:  function READ(offset, count, isPersistent)
2:      if not isPersistent then
3:          wait until (offset, offset+count) ∈ valid ∨ there is no persist instance
4:          if offset ∈ valid then
5:              return buffer[offset] – bufer[MIN(offset+count, buffer size)]
6:          end if
7:      end if
8:  end function
9:  function WRITE(offset, count, isPersistent)
10:     if not isPersistent then
11:         if no persistent writer instance then
12:             write to buffer
13:             valid = valid ∪ (offset, offset+count)
14:         else
15:             perform copy-on-write
16:             write to buffer copy
17:             validCopy = validCopy ∪ (offset, offset+count)
18:         end if
19:     else
20:         if (offset, offset+count) ∉ valid then
21:             write to buffer
22:             valid = valid ∪ (offset, offset+count)
23:         end if
24:     end if
25: end function
```

```
std::vector<LVFS::conf::Schema> buildSchema(){
  LVFS::conf::Schema s("Syslog", "Log");
  LVFS::LogModule::buildSchema(s);
  using attr = LVFS::conf::Attribute;
  s.addGlobal(attr::make_req("level", "string", "log level")
     );
  s.addGlobal(attr::make_opt("syslogLevel", "string", "Level
      to report to syslog", "info"));
  return { s };
}
```

Listing 5.13: Plugin Schema definition

of C functions. As mentioned in previous sections, LVFS plugins categories are Logging, Directory, Database, Cache, Content, Background, and Filters. We will discuss how each API for each category needs to be implemented.

5.2.1 Plugin schemas

As briefly mentioned in in section 5.1.8 LVFS will perform schema validation for plugins developed as part of LVFS core code and third party plugins. Plugins need to let the LVFS validator know what options they have, if those options are mandatory, what the value types are, etc.

Listing 5.13 shows the function which has to be implemented by the plugin. In this example, on lines 2 and 3, the `Syslog` plugin first creates a *schema object* and initializes it with options from the base Logging class. Lines 5 and 6 create a required option and an optional one. The arguments to the functions are the name of the attribute, the attribute type, and a description. Optional arguments also take an argument to define the default value.

A plugins schema can consist of multiple schemas which act as alternate versions. Listing 5.14 shows a stripped down example of a database plugin which has an optional attribute, *maxConnections*, and two alternate ways to establish a connection; (1) using username/password (2) using a raw connection string passed directly to the database API. First on line 2 we create a basic schema and add the optional option for *maxConnections*. Next we make a copy of this schema on line 3. Line 4 we define a required attribute to

```
std::vector<LVFS::conf::Schema> buildSchema(){
  LVFS::conf::Schema individual("PostgresDB", "Database");
  individual.addGlobal(attr::make_opt("maxConnections", "int
     ", "Maximum number of connections to keep open", "5"));
  LVFS::conf::Schema connectString(individual);
  connectString.addGlobal(attr::make_req("connectString", "
     string", "Raw connect string to pass to postgres module
     "));
  individual.addGlobal(attr::make_req("username", "string",
     "user used to connect to server"));
  individual.addGlobal(attr::make_req("password", "string",
     "password used to connect to server"));
  return { individual, connectString };
}
```

Listing 5.14: Plugin with alternate schema

specify the raw connection string to the new copy of the schema. On lines 5 and 6 we add the required attributes for username/password to the original schema. We then return both schemas. LVFS and the schema validator will ensure that the options specified in the configuration file will match at least one of the two schemas.

5.2.2 Common plugins

While each plugin has a specific set of API calls they need to implement, there are some common API calls for all plugins. Some plugin calls are optional and others are mandatory.

Since LVFS is primarily written in C++ we use inheritance and virtual functions to implement LVFS plugins. All plugins are derived from the LVFS Module class. Listing 5.15 shows the basic API common to all LVFS plugins.

All plugins can implement statistics reporting data as shown in Listing 5.15. In order to do so, plugins need to implement four API calls. If they chose not to, default calls will be used instead. The first of the four functions is for the plugin to return the names of all the different statistics it collects as a C++ vector of strings. For example, the database plugin could keep track of how many queries it has executed. One possible name it could give this statistic is *num_queries*. The next call is to specify what the units for each statistic name

```
class Module {
  virtual const std::vector<std::string>& statNames();
  virtual const std::vector<std::string>& statUnits();
  virtual const std::vector<std::string>& statValues();
  virtual const std::vector<int>&         statSlopes();
};
```

Listing 5.15: Basic plugin API

```
Module* createInstance(const conf::Module& conf);
Module* createInstance(const conf::Module& global, const
   conf::Module& local);
```

Listing 5.16: Basic plugin API

is. The third call returns the actual values for each statistic. If necessary, the plugin can reset the values after each call. The last call lets the statistics collecting system know if the values being returned are always increasing, decreasing, mixed, or unknown.

Besides the statistics collection functions there is a function that all modules need to implement. This function is used to create an instance of the plugin. In order to load plugins dynamically at runtime, LVFS utilizes the *dlopen* and *dlclose* POSIX function calls (The Open Group 2016b; The Open Group 2016a). Since these functions are C definitions, they are not able to load C++ classes. Each plugin, therefore implements a C based function for creating and destroying the C++ instances of the plugins. Listing 5.16 shows the function calls that should be implemented. The first implementation is for plugins which only exist *globally* and, therefore, only have global configuration settings. They are passed only a single argument which represents the global configuration options for that plugin. The second version is for plugins which have local *instances*. These plugins are provided with both *global* and *instance* configuration information. More information about *instances* is described in section 5.1.1. LVFS assumes all plugins are created using C++ based memory allocation and will take care of the destruction of those plugins using C++ based memory de-allocation on shutdown.

```
class LogModule : public Module{
  virtual void operator<<(logItem& output);
};
```

Listing 5.17: Logging plugin API

```
class DirectoryModule : public Module{
  int level;
  bool statByContent;
  virtual bool pathAccepted (const Path& path);
  virtual bool pathAttribute(const Path& path, struct stat*
     stbuf);
  virtual void pathListing  (const Path& path, DirEntry&
     entries);
};
```

Listing 5.18: Directory plugin API

5.2.3 Logging Plugins

Listing 5.17 shows the only function a plugin for logging needs to implement. The function takes a *logItem* object as argument. This object contains the string which needs to be logged. LVFS has already ensured that the log level of the message is of sufficient priority that it should be printed. The plugin simply needs to output the message to the plugin's log destination.

5.2.4 Directory Plugins

Listing 5.18 shows all the function calls a *Directory* plugin would implement to provide directory listings. The *pathAccepted* call provides an LVFS Path class. If the path at the same level as the level of the current plugin instance matches then the function returns true. This function is called by the tree traversal algorithm described in section 5.1.4 to determine which plugin a path belongs to. The *pathAttribute* function fills the POSIX `stat` attributes into the provided `stat` structure for the given path. This function will only be called if *pathAccepted()* would return true. The *pathListing* function provides a listing of files/directories for the plugin. The path is provided as a convenience if the plugin changes behavior based on the requested path. The resulting list of files/directories is stored in the

```
class DatabaseModule : public Module{
  virtual void dirQuery   (const std::size_t statementId,
     const std::vector<std::string>& args, const std::vector
     <type>& types, DirEntry* entries, bool directories,
     const Path& path) = 0;
  virtual void stringQuery(const std::size_t statementId,
     const std::vector<std::string>& args, const std::vector
     <type>& types, CacheInstance& cache, bool newLine,
     const Path& path) = 0;
};
```

Listing 5.19: Database plugin API

DirEntry class provided by LVFS.

5.2.5 Database Plugins

Database plugins are required to implement the two functions listed in listing 5.19. The first function definition performs a query, as defined by the *statementId*, and returns the result as a list of directory entries stored in the LVFS *DirEntry* class. If the query has place holders for arguments, these are passed to the functions as *args* and the argument types as *type*. The *directories* boolean specifies if the results of the query should be considered to be files (false) or directories (true). Finally, the input path is passed as the *path* argument in case the plugin changes behavior based on the input path.

The second function is similar to the first. The second function definition returns a the query results as a string with the columns and rows concatenated together. The results are stored into a cache instance. The *newline* argument specifies if a new line character should be added between each row in the query result.

5.2.6 Cache Plugins

A cache plugin consists of two components; (1) the *cache manager* class and (2) the *cache items*. We discuss in detail how *cache items* and *instances* work together in section 5.1.9. The *cache manager* is the class that manages when *cache items* expire and keeps track of all *items* managed by this cache. Listing 5.20 shows the function calls that need to be implemented as part of the cache manager component of the plugin.

```
class CacheModule : public Module{
  virtual CacheInstance get(const std::string& key);
  virtual CacheInstance add(const std::string& key);
  virtual void          del(const std::string& key);
  virtual std::vector<std::string> getOrphans();
};
```

Listing 5.20: Cache manager plugin API

```
class CacheModule : public Module{
class CacheItem{
  virtual ssize_t readInterface(char* buf, size_t count,
     off_t offset, bool persistent) const;
  virtual void writeModeChangedInteface();
  virtual void incrementWritersInterface();
  virtual ssize_t writeInterface(const char* buf, size_t
     count, off_t offset);
  virtual void truncateInterface(off_t length);
  virtual size_t availableInterface(bool persistent) const;
  virtual void releaseInterface();
  virtual void acquireInterface();
  virtual void attributeInterface(const std::string& name,
     const std::string& value);
  virtual std::string attributeInterface(const std::string&
     name) const;
  virtual void delAttributeInterface(const std::string& name
     );
  virtual void modifiedInterface();
};
```

Listing 5.21: Cache item plugin API

The *get* function call returns a *cache instance* to an existing *item* with the given key. If the cache contains no such key an uninitialized *cache instance* is returned. The *add* function will create a new *cache item* if it did not exist and return an *instance* to the *item*. If the *item* with the given key already existed, then the cache would return an *instance* to that *item*. With this design the *add* function will always return a valid *cache instance*. A cache plugin differentiates between existing *items* in cache and new *items* by setting the *cache instance* mode to *persistent* as described in section 5.1.9. A newly created *cache instance* is returned by the *add* function in *persistent* mode. Depending on the cache plugin implementation, a third mode could be an item which exists in the cache but has expired. In those circumstances, the cache can return an *instance* to the existing *item* with *persistent* mode set. The *del* function deletes the given key from the cache if it exists. The optional function *getOrphans* is called on LVFS startup to provide a list of existing *cache items*. This is only useful for cache plugins which persist information across runs of LVFS.

Listing 5.21 shows the API that a cache plugin needs to implemented to support *cache items*. The *readInterface*, *writeInterface*, and *truncateInterface* are the functions to read, write, and truncate data in the cache respectively. The *writeModeChangedInterface* is a convenience function which tells the plugin that the cache item has changed between *direct* and *copy* mode. As explained in more detail in section 5.1.9, *cache items* can enter copy-on-write mode. This function notifies plugins of the change and allows plugins to merge copies back when exiting *copy* mode. The *incrementWritersInterface* is a convenience function notifying the plugin when a *cache instance* to this *cache item* has switched to write mode. The *availableInterface* function returns the number of bytes currently stored in this *cache item*. The *releaseInterface* and *acquireInterface* are used to notify the plugin that currently there are no *instances* referencing this *item* or that at least one *instance* has re-acquired this *item*. The purpose of these functions is to allow the cache plugin to release any resources, such as file descriptors, when not necessary and re-acquire them when needed. The *attribute* interfaces are used for setting, reading, and deleting attributes. Finally, the *modifiedInterface* is a convenience function that tells the cache plugin when an *item* changes state between having local modifications and being up to date with the remote source.

```
class ContentModule : public Module{
  virtual std::vector<std::string> schemes() = 0;
  virtual bool writable() const;
  virtual void retrieve(const Path& path, std::shared_ptr<
     Location> location, CacheInstance& inst) = 0;
  virtual void stat(const Path& path, std::shared_ptr<
     Location> location, struct stat* stbuf) = 0;
  virtual void store(const Path& path, std::shared_ptr<
     Location> location, CacheInstance& item);
};
```

Listing 5.22: Content plugin API

5.2.7 Content Plugins

Content plugins, as described in section 5.1.5, are used for retrieving content from remote sources or storing them. The functions needed to implemented a content plugin are shown in listing 5.22. A content plugin is required to implement the *retrieve*, *stat*, and *schemes* functions while the others are optional. The *retrieve* plugin is for retrieving the contents of a file from the provided *location*. The plugin should store the content in the provided *cache instance*. The *stat* function retrieves POSIX `stat` information about the file from the remote source and populates the information in the POSIX `stat` struct provided. If a plugin has write support it will implement the *writable* function and return true. The plugin will also implement the *store* function. This function stores the contents of the file from the local *cache instance* back to the remote *location*.

5.2.8 Background Plugins

Background plugins are generic plugins that perform unknown tasks. Therefore, background plugins have no specific API needed. Any plugin which uses the *Module* class, described in section 5.2.2, as their base class is considered a background plugin. While a background plugin may implement the statistics function calls defined in the base class, they will never be called for background module.

```
class FilterModule : public Module{
  virtual Path getSourcePath(const Path& path) = 0;
  virtual bool pathAccepted(const Path& path, const DirEntry
     & entries) = 0;
  virtual bool pathAttribute(const Path& path, const
     DirEntry& entries, struct stat* stbuf) = 0;
  virtual void pathListing(const Path& path, const DirEntry&
      entries, DirEntry& newEntries) = 0;
  virtual void pathOpen(const Path& path, CacheInstance
     inputContent, CacheInstance& outputContent);
};
```

Listing 5.23: Filter plugin API

5.2.9 Filter Plugins

Filter plugins are responsible for dynamic content conversion. They work nearly identical to directory plugins and content plugins together. Listing 5.23 shows the function calls a filter plugin needs to implement. The functions *pathAccepted, pathAttribute, pathListing* behave the same way as their directory module counterparts described in section 5.18. The main difference is that each function is provided with an additional argument, *entries*. This argument represents the file/directory entries generated by the directory plugin which this plugin is filtering. For example, the *pathListing* function returns new file/directory entries as the *newEntries* argument generated from the file/directory listing specified in *entries*. The *getSourcePath* function lets the plugin tell the caller what the original path is for a path passed in as the argument. A **JPG** to **PNG** filter plugin, for example, would return the original **JPG** file for the provided **PNG** path. The *pathOpen* plugin generates a new *cache instance* which contains the filtered output from the *cache instance* passed which represents the original source content.

Chapter 6

CASE STUDY: MODAPS DATA DISTRIBUTION TREE

Below we describe how LVFS is being used by MODAPS to distribute data via the Level-1 and Atmospheres Archive & Distribution System (LAADS). MODAPS and LAADS are two projects that run at the Terrestrial Information System Lab's data center at NASA Goddard Space Flight Center. MODAPS is a data processing system responsible for receiving raw satellite data, called level 0 data, and process the information into higher level products more convenient for scientific research. The data produced is distributed by LAADS and other places. In addition to data produced by MODAPS, LAADS distributes data by other processing systems. In this chapter we will look at the unique problems associated with producing and distributing data with MODAPS/LAADS and how LVFS helps overcome those problems.

6.1 Problem Description

Scientific data sets, such as the Large Hadron Collider, Sloan Digital Sky Survey, and Brain fMRIs, are growing constantly and producing petabytes of data with some data records doubling every year (Szalay & Gray 2006). Managing petabytes of data in billions of files present complex problems. MODAPS contains approximately 40 petabytes of data in over 2.2 billion files. In order to ensure quick processing MODAPS stores data on drives in a layout which is convenient for the processing system. This ensures that any processing can be performed as quickly as possible.

LAADS distributes data using many different protocols including HTTP and FTP. In order to distribute the data, LAADS would NFS mount all remote disks on the HTTP and FTP servers. It would then generate a directory tree consisting of symbolic links to the files

stored on the NFS mounts. This directory tree structure was maintained by a script which would compare the information in the LAADS metadata database against the symbolic links on the local server. Whenever new files were ready for distribution MODAPS would insert entries in the LAADS metadata server and this script would generate the necessary new links to the NFS mounted disks. Additionally, this script would re-check existing links and ensure they matched information in the metadata server.

The problem with this design is the lack of scalability and flexibility. As the amount of data and the number of files was increasing, the amount of time required to maintain the symbolic links would increase linearly taking over a week for the script to perform one iteration. This meant that at times it would take nearly a week for LAADS to make files available. Additionally, with this design, if LAADS wished to distribute data in an alternate directory layout it would mean running a second script to generate the new symbolic links which would also grow linearly with respect to the file count in the metadata server.

6.2 LVFS Implementation

To switch LAADS to LVFS we need to identify the directory layout and how to obtain the information necessary to create the directory layout. Since LAADS only distributes data, this case study involves a read only configuration of LVFS. The data tree distributed by LAADS is structured at the top level to be a list of all supported versions of the data, followed by the different data products available for a given version, followed by the years and day of the year (doy) there is data available for the given product. An example path would be */6/MOD04_L2/2017/012/*. The top level directory is specifying that we are looking for version 6 data. The product's name is MOD04_L2 and we are interested in data from day 12 of 2017.

To generate this directory structure in LVFS we need 6 levels and, therefore, 6 queries. These would be:

- List of all available versions
- List of all products given version
- List of all years given a product and a version
- List of all days given a year, product, and version

- List of all files given a day, year, product and version
- Location of a file given a file name

```
1  - type: SQLDir
2    query:
3      statement: select "ArchiveSet" as name, 1 as size, extract(epoch
           from current_timestamp) as time from "ArchiveSet_Def" where "
           IsPublic" = true
4    children:
5      - type: SQLDir
6        query:
7          statement: select "ESDT" as name, 1 as size, extract(epoch from
               current_timestamp) as time from "ArchiveSet_ESDT_Def" where
               "IsPublic" = true and "ArchiveSet" = $1
8          bindings: [ "${-1}" ]
9        children:
10         - type: SQLDir
11           query:
12             statement: select distinct(to_char("DataDate", 'YYYY')) as
                   name, 1 as size, extract(epoch from current_timestamp)
                   as time from "DataAvailability" where "ArchiveSet"=$1
                   and "ESDT"=$2
13             bindings: [ "${-2}", "${-1}" ]
14           children:
15             - type: SQLDir
16               query:
17                 statement: select distinct(to_char("DataDate", 'DDD'))
                       as name, "DataCount" as size, extract(epoch from
                       current_timestamp) as time from "DataAvailability"
                       where "ArchiveSet"=$1 and "ESDT"=$2 and "DataDate"
                       >= $3::date and "DataDate" < $4::date + interval '1
                       year'
18                 bindings: [ "${-3}", "${-2}", "${-1}-01-01", "$
                       {-1}-01-01" ]
19               children:
20                 - type: SQLFile
21                   query:
22                     statement: select "FileName" as name, "FileSizeBytes
                           " as size, extract(epoch from "OnDiskTime") as
```

```
                    time from "getFM_ArchiveDir"($1, $2, $3::
                    timestamp + $4::interval - '1 day'::interval, $5
                    ::timestamp + $6::interval)
23                bindings: [ "${-3}", "${-4}", "${-2}-01-01", "${-1}
                    day", "${-2}-01-01", "${-1} day"]
24              locator:
25                deferred: true
26                location:
27                  db: PostgresDB
28                  statement: select 'http://' || "Host" || ':8080/f'
                        || "Disks"."DiskId" || '/lads/archive/' || $1
                        || '/' || $2 from "Disks", "FilesOnDisk", "
                      File" where "File"."FileName"=$3 and "
                      FilesOnDisk"."FileId"="File"."FileId" and "
                      Disks"."DiskId"="FilesOnDisk"."DiskId"
29                bindings: [ "${-3}", "${0}", "${0}"]
```

Listing 6.1: LVFS Configuration for LAADS distribution

Listing 6.1 shows the YAML configuration file which generates the directory tree for LAADS. Each level consists of an *SQLDir* plugin which generates a directory listing based on an SQL query. The last level is an *SQLFile* plugin which generates the file listings. Line 3 defines the query to be executed to find all supported versions of data sets. When a directory listing for */* is requested, LVFS will perform this query, if not already cached, and return the results as a list of directories. Assuming the directory listing contains an entry named *6* and a directory listing of */6/* is requested then LVFS will traverse directory tree, as described in section 5.1.4, to arrive at the second level show on lines 5–9. LVFS, via the SQLDir plugin, will perform the query defined on line 7 to obtain a new listing. This time, however, the query depends on the parent directory as can be seen from the inclusion of query bindings on line 8. This binding tells the SQLDir plugin to use proper escaping and replace the *$1* variable in the query with the value of *$-1*. As described in section 5.1.4, this value represents the parent directory that led to the current directory which, in this case is the string *6*. This way LVFS has now performed a query to find all version 6 products. This continues all the way to the last level where LVFS uses the SQLFile plugin to generate a list of files.

Lines 24–29 are used when a file is opened. The exact behavior is described in section 5.1.5. In this example, the LAADS database maintains all the necessary information to

locate a file including the host, the disk on that host, and the path on that disk. Using this information we create a deferred locator. As described in section 5.1.5, a deferred locator means that LVFS should use the result of the provided information as the location. In our case the provided information is a database query whose result is an http location to the file content.

The part not shown in listing 6.1 is all the caching mechanisms. In this use case, LVFS was setup with a memory based cache plugin for short term caching, and a disk based plugin for long term caching. Because of this caching, the steps above will only be taken on some occasions. Each time the a query is performed LVFS will store the results in either long term cache or short term cache so that subsequent accesses will skip all the database queries and provide the answers immediately.

6.3 Performance

We tested the performance of LVFS against the original NFS system in both a simulated and production environment. Since LAADS is only providing read access both the NFS mounts and LVFS were configured with read-only access. We will first look at the simulated environment and then the production environment.

For the simulated environment we used a test setup consisting of two computers. Table 6.1 shows the hardware specifications of the two computers. The server was used to serve the files once configured for NFS and then configured for LVFS. The files served out consisted of eight 128MB files totalling 1GB of data read by the client. When using LVFS, the server was setup with the Nginx HTTP server for CentOS 7. We timed the amount of time it takes to read all eight files. Each round used a different block size for reading and different number of threads. After each round all caches were emptied to ensure no local access was skewing the results.

Figure 6.1 shows the results of the runs for one, two, four, or eight threads reading between 512 and 8192 bytes at a time. We can see that at 512 and 1024 bytes LVFS is slower than NFS. This is due to the overhead of LVFS running in user space. All POSIX request for LVFS enter kernel space and exit into another user space application while with NFS the requests are processed in kernel space. As the block size is increased, the number of POSIX calls is reduced and, therefore, the amount of overhead for each request becomes less prominent. At 2048 byte block sizes the overhead for running in user space is no longer

	Server	**Client**
Network	1Gb/s	1Gb/s
RAM	128GB	48GB
OS	CentOS7	CentOS 7
CPU	Xeon 3.4GHz	Xeon 2.8GHz

Table 6.1: Hardware Specifications of the testbed system for LVFS read benchmark

measurable. Similarly, we can see that LVFS is capable of handling simultaneous requests just as well as NFS but with the addition of new features such as content verification.

We also ran experiments in a production environment to test out LVFS' performance in real world conditions. Figure 6.2 shows the monthly data volume for the LAADS FTP server. Between July and August LVFS replaces the NFS based system. We can see the data download volume of LAADS nearly double from 90TB a month to approximately 180TB a month.

The use of a script to generate the organized directory structure, as described in section 6.1 was the main result of increase in performance. Because the symbolic generation and verification took over one week to execute one full iteration, the LAADS team could not load balance the FTP server. Running two FTP servers meant having a script on each computer. This could result in the directory structure of each server being up to a week off. For a project which generates new datasets every few minutes this would result in an inconsistent FTP directory structure with files appearing and disappearing based on which server a user's connection happens to hit. Since LVFS generates the directory structure on demand from the LAADS metadata server, the LAADS project can now install LVFS on as many FTP servers as needed to support user demand.

Figure 6.3 shows the resource requirements of LVFS during various levels of load. For the same time period we can see in Figure 6.3a that LVFS at its peak maintains 220 actively open files while Figure 6.3b shows it utilizing approximately 20 Gigabytes of swap space for those files. During the entire period Figure 6.3c shows that the CPU is nearly fully idle.

Finally, we measured detailed CPU statistics of the LAADS three load balanced HTTP servers distributing over 20 Petabytes of data in over 2 billion files. Figure6.4 shows the combined data rate of the three servers for a one week period from April 27th, 2017 to July 4th, 2017. On average, LAADS was distributing data at approximately 3 Gigabits/sec.

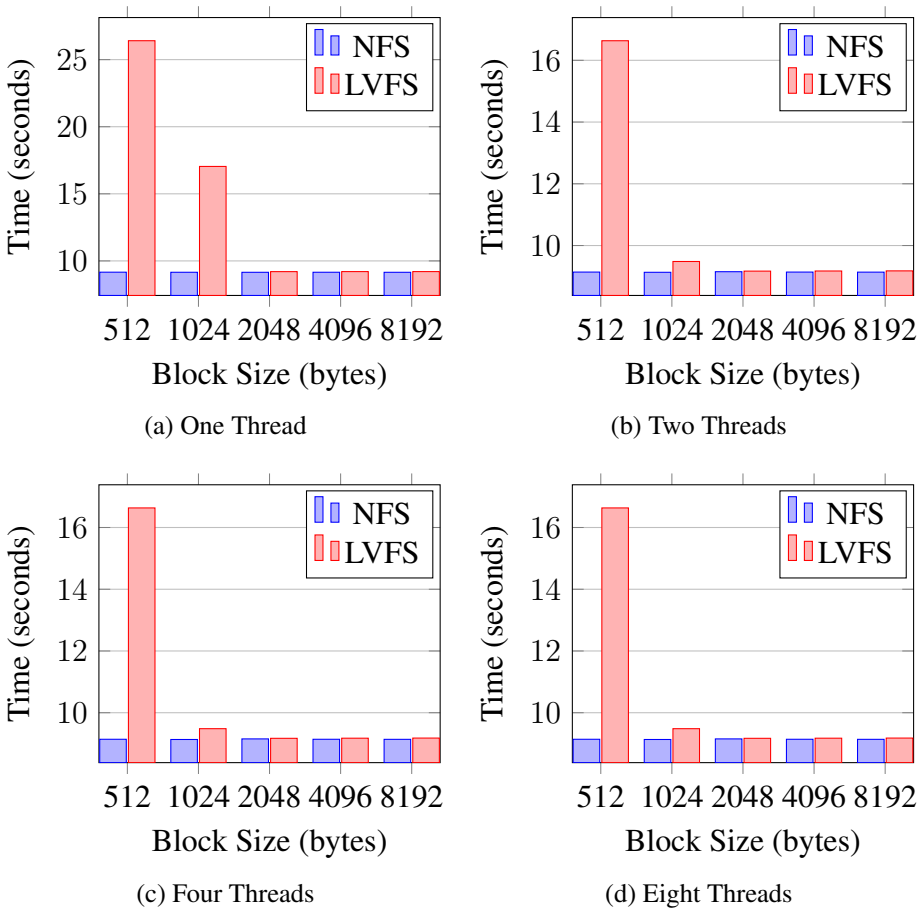


Figure 6.1: Read time comparison between LVFS and NFS with read blocks between 512 and 1024 bytes and between 1 – 8 simultaneous readers

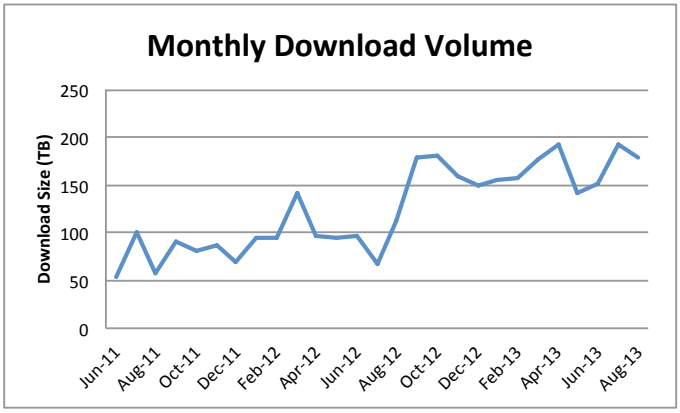
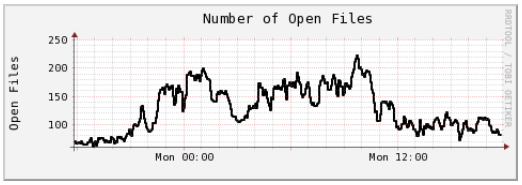
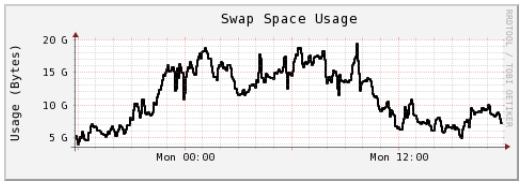


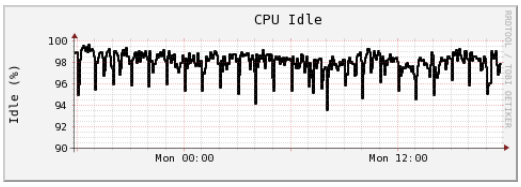
Figure 6.2: Monthly FTP download volume in 2011, 2012, and 2013



(a) Number of open files



(b) LVFS used Swap Space



(c) CPU Load

Figure 6.3: Resource usage by LVFS for a one day period on the LAADS FTP Server

Figure 6.5 show detailed CPU usage of the three load balanced servers. We can see that the computers are nearly idle just like the previous example. Additionally, we can see that the I/O wait is at 0% meaning that LVFS was capable of processing POSIX requests fast enough so that the CPU was never waiting on I/O.

6.4 Conclusion

LVFS has proven to be an invaluable tool for distributing data for LAADS in a flexible manner. Being able to restructure entire dataset on the fly with no downtime or the ability to stand up new hosts that appear to have all of the data available locally in minutes is very beneficial. Not only has LVFS been proven to be beneficial due to its flexibility but benchmark results have shown that it is capable of making Petabyte scale datasets available on a single node while allowing hundreds of processes to access the datasets simultaneously with minimal load on the system. Benchmarks have shown that we can double data throughput from LAADS' public FTP server due to the ability to easily create load balanced server. We believe LVFS can scale to exabyte sizes without modification due to the fact that it currently manages Petabyte scales with minimal load. LVFS is only limited by the scalability of the metadata server which can be replaced with better performing systems without any modification to LVFS.

Additionally, since LVFS makes use of the highly detailed existing metadata informa-

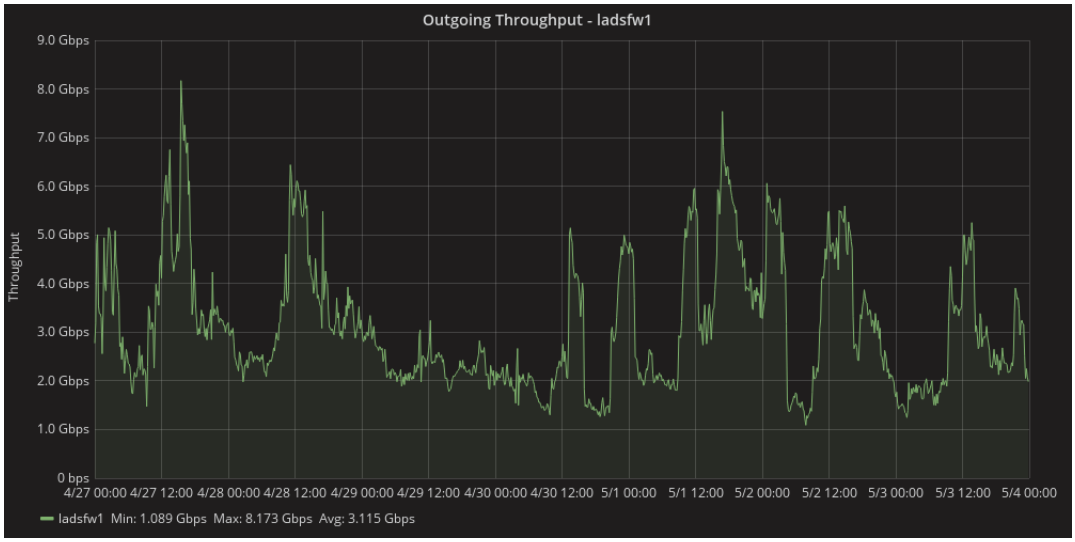
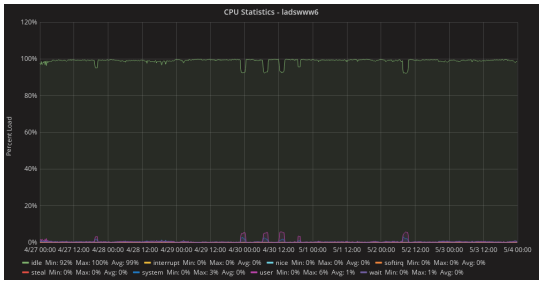
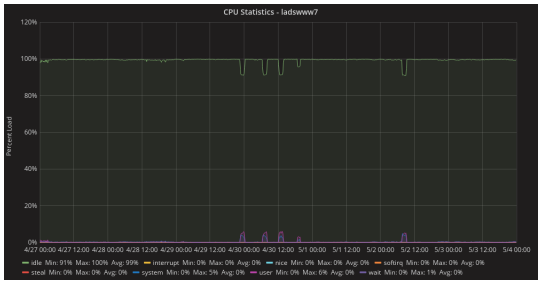


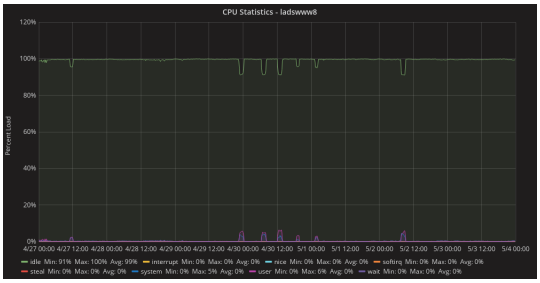
Figure 6.4: Combined data transmission rate of three load balanced web servers for LAADS



(a) First web server



(b) Second web server



(c) Third web server

Figure 6.5: CPU usage of three load balanced web servers distributing LAADS data

tion present in most scientific processing system, LVFS has the unique ability to describe the same dataset in multiple views without requiring extra information or duplicating information. This gives LVFS the ability to organize the same dataset in a directory structure by measurement date, measurement region, or any other aspect relevant to the scientific dataset.

Chapter 7

CASE STUDY: WRITE SUPPORT

The first case study concentrated on read-only support for distributing processed data via LAADS to the public. While that setup is good enough if there are no plans to change the underlying storage architecture, it will become insufficient if the project or the data center decides to transition to new storage architectures. With existing file system designs, when new storage concepts are introduced it requires modification of the project code and re-training of developers to learn the new storage concepts. For this case study we will look at how a project can add new storage architectures to the system without requiring a complete redesign of their code.

We will use the same real world example of LAADS for this study. Since MODAPS/LAADS has been running for decades the code primarily uses concepts of block based storage. It is written to work with files and directories and uses POSIX calls However, now that object based storage systems have matured and provide new features, such as handling hundreds of billions of files and providing better redundancy, it is would be desirable to add such storage architecture to the project. In this case study we will look at the problem of incorporating a new storage architecture and how the MODAPS/LAADS project addressed it via LVFS.

Switching architectures after supporting an older one for decades is a complicated task. Over time a project's code tends to grow to add new features, handle extreme cases, and requires changes to handle the growth of the project. As a result switching or adding new architectures to a long running project can be very time consuming or even impossible. In the case of MODAPS/LAADS this complication comes from decades of using block based storage and transitioning to more modern storage architectures. Rewriting decades of code to support a new storage architecture would require months of code rewrite and

months of testing to support.

While some projects might decide spending many months to add support for a new storage protocol acceptable, this becomes more complicated due to the fact that the object store environment is still in its infancy. As a result, there are many competing protocols in the wild. The HGST systems utilize Amazon's S3 protocol but DDN's WOS and Seagate's Kinetic drives utilize their own protocols. Clearly supporting all of these protocols would be very costly in terms of time and cost spent to add support for each protocol.

As an alternative solution, a project could decide to not support all protocols but only utilize one. Depending on the size of the project, this could be a viable solution. If the project is only a small project in the data center they are not likely to have much influence on the data center's architecture decision. Even for a large project this would not be an easy decision. As the object store environment matures the decisions made early on might end up being the wrong ones and the competing protocols might win over the protocol selected.

Finally, as was the case with the Terrestrial Information Systems Lab, the purchase of HGST object store drives was part of a bigger plan to create a hybrid storage architecture to try out different architectures and evaluate which are better and to create a hybrid storage architecture to better insulate against vendor lock in. Using a hybrid storage architecture would have been impossible to implement if it would have required all projects within the Terrestrial Information Systems Lab to implement the code changes for each new architecture.

7.1 Problem Description

In this scenario, the Terrestrial Information Systems Lab has approximately 3 Petabytes of new storage space in the form of HGST's Active Archive System. This storage utilizes Amazon's S3 protocol to expose the storage as an object store. The goal is to take advantage of this new storage architecture and the new features it offers such as object based access, erasure encoding, and high bandwidth for simultaneous access.

With standard storage methods utilizing this new storage architecture would require one of three possible options. The projects wishing to utilize this storage would have to develop their own implementation of the S3 protocol, incorporate third party libraries which implement the S3 protocol into their code, or utilize third party applications outside of their coding environment.

The first two options are not desirable because they require significant code changes to projects. For decades projects running at the Terrestrial Information Systems Lab have used block based systems as their underlying storage. Therefore, all code has been developed relying on standard POSIX calls and directory structures. The addition of a new concept, such as object based storage, would require major changes to code to support. Additionally, since object based storage is not fully replacing the vast block based architecture in play, the old POSIX based code could not be abandoned either. Implementing the first option would require a lot of code development and maintenance. Implementing option two would only be feasible if libraries for all languages used were available and stable.

The third option would address the language compatibility problems but would still suffer from increasing code complexity to use different code execution paths depending on where data was being read from or written to. Further, the use of third party executables would add additional overhead since the executables would not be running in the same process environment as the project code. This would complicate issues in that the project code would have to spawn a new process to copy the files from the remote S3 storage to local storage, process the input, produce new output or modify the existing input, and finally transfer the new output and potentially the input back to S3 storage or other storage.

In addition to code complexity for the project comes the transition strategy for the data. Once a dataset has been identified to be moved to the new storage all new data belonging to that dataset needs to be stored on the new storage. However, the old data will continue to reside on the old storage architecture and be slowly moved to the new storage. The project would require to maintain some sort of table to identify where each file resides.

In section 7.2 we will discuss how such a transition was done using LVFS and how using LVFS avoided all the problems described above. We will then look at the performance of transitioning to the processing system and processing new files to the new architecture in section 7.3. We will describe our conclusions in section 7.4.

7.2 LVFS Implementation

Figure 7.1 shows the desired infrastructure layout at the Terrestrial Information Systems Lab after adding S3 storage. Multiple block based storage systems exist and contain the existing products. The new S3 storage has been added and all new files of certain products should be stored in the S3 storage system. The distribution systems should retrieve

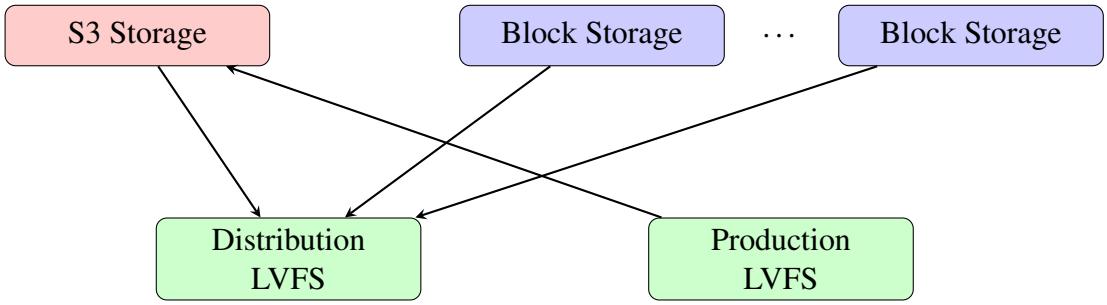


Figure 7.1: Terrestrial Information Systems Lab infrastructure after addition of S3 storage

files from the block based or S3 storage. The products identified for moving to the S3 storage were all products based on the MERIS and Sentinel instruments. In addition to moving new files from those instruments to the S3, existing files already stored on the block based storage should be moved to the S3 storage as well.

The implementation of the S3 storage architecture was done in two stages. The first stage was to develop a content plugin for LVFS which communicated over the S3 protocol. The next was to create an LVFS configuration which allowed for a slow transition of products in LAADS to the S3 storage without any downtime or any changes to existing MODAPS/LAADS code.

As described in section 5.2.7, a content plugin needs to implement between three and five functions in order to function correctly. If the content plugin only provides read support it only needs to create the three functions *schemes, retrieve,* and *stat*. However, since MODAPS will be both writing and reading from S3 storage seamlessly we will need to additionally implement the *writable* and *store* functions.

The *schemes* plugin specifies which string, called a scheme, at the beginning of the URI will be assigned to this plugin. In this case we chose the string *s3* so that any URI beginning with *s3://* will be handled by this content plugin. We utilized the libs3 (bji 2017) open source library to provide the S3 communication protocol. The implementation of the remaining of the S3 plugin consisted of translating the LVFS calls into libs3 calls and vice versa. Overall the plugin took approximately one week to write and debug and consisted of 266 lines of code including formatting and comments.

We used two different configurations for LVFS. One configuration was used for the production system with write support and the other was used for the distribution system with read only support for additional security. Listing 7.1 shows a partial configuration

file for LVFS on the production machines which will be storing new Sentinel and MERIS products. Lines 1–3 and 9–11 create a total of 6 directories at the top level. These directories are the names of the different MERIS and Sentinel products like *MER_FRS_1P*. Lines 2 and 10 setup the ACL permissions which specify that the unix group *prodGroup* will be allowed to write to these directories and everyone else will only be allowed to read. Lines 5–8 and 13–16 define the LVFS settings to let LVFS know where to retrieve and store files. Since we wish to store files on the S3 storage the locator is defined with the s3 scheme on line 8 and 16. The format of the URI is *s3://<host>/<bucket>/<file>*. In our case the bucket is a unique number which represents a versioning information used by LAADS for distribution followed by the product name. Because the product names used by MODAPS for production are not compatible with S3 bucket names we use LVFS' Lua functionality to transform the product name into an S3 bucket with valid characters. In this case we replace the _ character with - and we convert all upper case characters to lower case. With this configuration the production system can store Sentinel and MERIS products into LVFS simply by using the unix *cp* utility to copy the files into the LVFS path or by directly creating the files inside the LVFS path.

```
1  - type: Dir
2    acls: [ 'g:prodGroup:rwx' ]
3    name: [ MER_FRS_1P, MER_FRS_BP, MER_RR__1P, MER_RR__2P ]
4    children:
5      - type: File
6        allowAll: true
7        acls: [ 'g:prodGroup:rwx' ]
8        locator: 's3://10.0.0.${random(0,2)}/491-${to_lower(replace(path
             [1], "_", "-"))}/${0}'
9  - type: Dir
10   acls: [ 'g:prodGroup:rwx' ]
11   name: [ S3A_OL_1_EFR, S3A_OL_1_ERR, S3A_SL_1_RBT ]
12   children:
13     - type: File
14       allowAll: true
15       acls: [ 'g:prodGroup:rwx' ]
16       locator: 's3://10.0.0.${random(0,2)}/450-${to_lower(replace(path
             [1], "_", "-"))}/${0}'
```

Listing 7.1: LVFS Configuration storing MERIS and Sentinel data

Listing 7.2 shows the partial configuration for LVFS to read Sentinel and MERIS data from either block or S3 storage. This configuration is appended to the configuration in Listing 6.1 after line 29. This new configuration change utilizes LVFS *any* functionality described in section 5.1.5. The first change is to apply a condition to the original file listings. This condition checks if the path includes a the number *450* or *491* which represent products for MERIS or Sentinel. If the path includes any other value the original configuration for LVFS is used. If, however, the path includes either of those values then LVFS will skip the original configuration from lines 20–29 and use the configuration from lines 2 – 12 from Listing 7.2. This configuration is nearly identical as the original one. The exception is that this configuration uses two locators with the *any* specifier. This configuration tells LVFS to use two possible locations for MERIS and Sentinel products. The first location is identical to the location pattern used for all other products. The second, alternate, location is the URI for the S3 storage. LVFS will first attempt the default block based storage for files and, if those fail, will fall back to S3. Once more than half of the products are located on S3 storage there would be a small performance gain to swap the two locators so that LVFS will attempt a download from S3 first. This method is not the only way to configure LVFS. A less verbose but slightly less efficient way would be to use the original configuration and always use the two locators with the *any* specifier. The results would be the same but for MERIS and Sentinel products LVFS would always first attempt file retrieval from block storage first before falling back to S3.

```
1    condition: ( "${3}" ~= "450" and "${3}" ~= "491" )
2  - type: SQLFile
3    query:
4      statement: select "FileName" as name, "FileSizeBytes" as size,
           extract(epoch from "OnDiskTime") as time from "getFM_ArchiveDir
           "($1, $2, $3::timestamp + $4::interval - '1 day'::interval, $5::
           timestamp + $6::interval)
5      bindings: [ "${-3}", "${-4}", "${-2}-01-01", "${-1} day", "$
           {-2}-01-01", "${-1} day"]
6    locator:
7      any:
8        - deferred: true
9          db: PostgresDB
10         statement: select 'http://' || "Host" || ':8080/f' || "Disks"."
               DiskId" || '/lads/archive/' || $1 || '/' || $2 from "Disks",
```

```
              "FilesOnDisk", "File" where "File"."FileName"=$3 and "
             FilesOnDisk"."FileId"="File"."FileId" and "Disks"."DiskId"="
             FilesOnDisk"."DiskId"
11         bindings: [ "${-3}", "${0}", "${0}"]
12       - s3://10.0.0.${random(0,2)}/${-4}-${to_lower(replace(path[3], "_
           ", "-"))}/${0}
```

Listing 7.2: LVFS Configuration retrieving MERIS and Sentinel data

The above configurations setup LVFS for the long term solutions of putting new products into S3 and retrieving products from S3 and block storage. Once all products have been transferred to block storage the LVFS configuration can be simplified by removing the block storage locator (lines 7–11 from Listing 7.2) definition. The final aspect is to write a script to transfer existing products from block storage to S3. A simple way would be to write a script which reads the files from LVFS and write back to LVFS. A temporary host was setup with an LVFS configuration similar to Listing 6.1. The configuration in Listing 7.1 was prepended to that configuration. A simple recursive copy can now transfer existing products into S3.

7.3 Performance

We tested LVFS performance by measuring the network throughput, memory usage, CPU usage, and the LVFS backlog while transferring files from block storage to S3. The hardware used was the same as the one for the use case example in section 6 and is described in Table 6.1.

Figure 7.2a shows the network data rate for incoming and outgoing data on the host performing the reads and writes. The graph covers a 24 hour period while files were being copied from block storage to S3. From the graph we can see LVFS was able to hold a consistent average receive rate of approximately 750 Mbps. Simultaneously, LVFS was holding an average transmit rate of 750 Mbps. We can see that there was a greater variance in transmission rates than read rates. This is due to the fact that LVFS will transmit data only in consistent objects to the S3 storage. Because the block storage system is slower than the new S3 storage, LVFS would read at a consistent rate from block storage but then would send the data in faster bursts to S3 followed by periods of less activity while it waited for more data from the block storage. Additionally, we can see that transmission rate would

achieve and briefly exceed the advertised transmission rate of the network card.

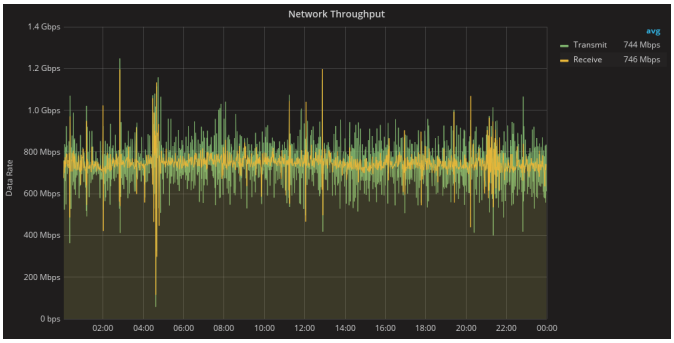
Figure 7.2b shows the CPU load for the same 24 hour time period. The average 1, 5, and 15 minute loads of the system are around 0.3%. Similarly, Figure 7.2c shows that LVFS is utilizing nearly 100% of the 128GB of RAM available. However, LVFS is utilizing it only as cached memory. Should any other process require memory the Linux kernel would re-allocate the desired memory to other processes. In other words, LVFS is utilizing as much unused memory as possible without affecting the memory requirements of other processes. Finally, Figure 7.2d shows the on-disk cache used by LVFS to store copies of the files being transitioned from block to S3 storage for the 24 hour period. LVFS was configured to use up to 80% of the disk space assigned as cache. We can see that LVFS was utilizing as much cache as possible and when exceeding the 80% barrier LVFS would reduce its usage to 70%.

Figure 7.2e shows what we call the LVFS backlog. These are files that have been flagged as having local modifications which either are in the process of being transferred to the remote storage or will be transferred once there are no local clients making modifications to the file. The figure shows that, on average, five files had local modifications. Since there were four threads reading files from the block storage that means on average LVFS was only transferring a single file to S3 and waiting for four files to finish being read from block storage. This confirms as well that the S3 storage is faster than the block storage. The graph also shows occasionally the backlog increasing to nine or ten files. This is due to all threads sometimes finishing reading files from block storage simultaneously while LVFS is busy with uploading another set of files. We can see from the quick drops that LVFS quickly catches up with the backlog and the numbers return to four or five files.

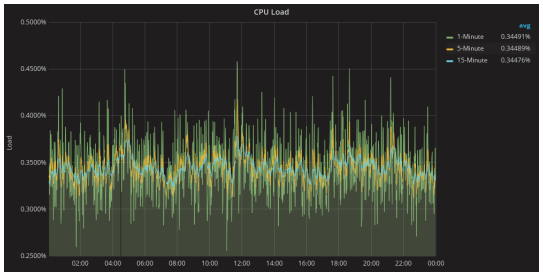
7.4 Conclusion

From the performance results we can see that LVFS write support is more than capable of keeping up with the read support. The previous use case showed that LVFS is capable of performing reads as fast as NFS as long as reasonable block sizes are used to avoid the overhead of running in user space. While LVFS is just as fast as other storage systems it performs automatic checksumming to ensure no data corruption has happened during network transfer.

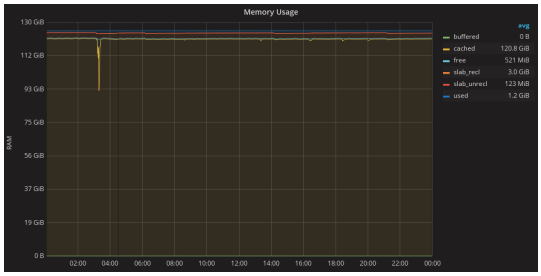
While no loss in performance is nice in addition to better protection against data cor-



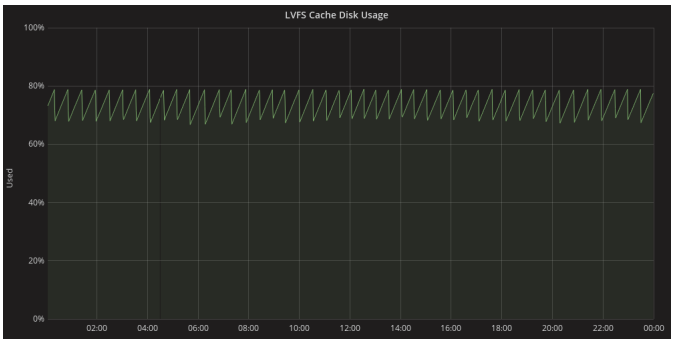
(a) Network Throughput



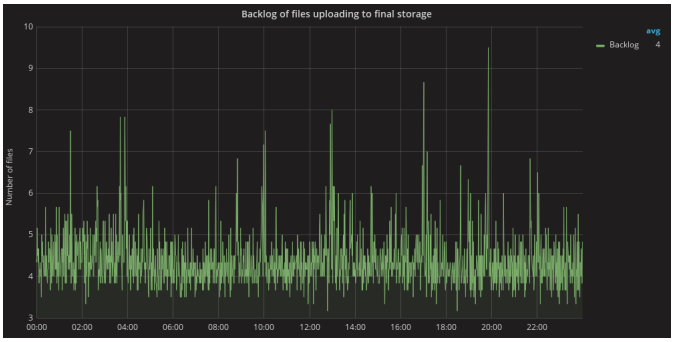
(b) CPU Usage



(c) Memory Usage



(d) Cache Usage



(e) Backlog

Figure 7.2: Network, CPU, Memory, Cache, and Backlog metrics during transfer of files from block storage to S3

ruption, the real strength of LVFS is the ability to provide the necessary flexibility to add storage system with brand new concepts designed decades after a project's inception without requiring any changes in the project's code. We showed a use case where S3 storage was added to an existing project without making any changes to the project's code. In addition, existing files were transferred to the S3 storage from the block storage with minimal effort and no outage time. All of this was achieved without the loss of performance.

Chapter 8

CASE STUDY: ON-DRIVE MAPREDUCE

The increase in processor performance, the decrease in processor size and power requirement combined with the increase in data intensive computing has created a push to move computation as close to data as possible. This push was predicted by many researchers as early as the 1990's (Acharya, Uysal, & Saltz 1998; Riedel, Gibson, & Faloutsos 1998). While CPU technology needed to catch up, methods such as the MapReduce programming model were developed to move computation to computers closer to the data (Dean & Ghemawat 2004; Zhao & Pjesivac-Grbovic 2009). On top of this, data centers have been running into internal network bottlenecks and network congestion. The computation power within data centers is often sufficient for the tasks at hand, but these computation resources can't be fed with data fast enough because of the network congestion bottleneck. Thus, the logical step to this problem is to either increase network speeds or reduce the amount of data communicated over the network. Increasing network speeds is often associated directly with an increase in total storage system costs, as storage servers need to either upgrade to new, faster, and more expensive hardware, or they have to build their storage systems with those more expensive networks in the first place. For the purpose of helping to reduce storage system costs for clients, and to help to alleviate the network speed bottleneck, Active Drive focuses on the latter of the two solutions, via reduction or filtering of the data before it is sent out on the network. In this paper we explore the next logical step in this evolution in computing; moving computation directly to storage. We will show how to utilize Seagate's Active Drives to perform general purpose parallel computing using the same MapReduce programming model developed by Google.

The MapReduce programming model facilitates parallel programming and computation close to data by borrowing two concepts from functional programming: the Map

function, which processes input data and produces key/value pairs as output, and the Reduce function, which processes the key/value pairs from the Map output and produces final key/value results as output. With this design many algorithms can be adapted to be run in the MapReduce programming model (Chu *et al.* 2006). Such parallelization achieves quicker computation than other methods by attempting to reduce the amount of data that is moved between computers.

In this case study we will expand LVFS to utilize the Seagate Active Drives for storage as well as enable the MapReduce programming model to perform general-purpose computing directly on a drive. We will show our MapReduce implementation on Seagate Active Drives, and we will show our results: a significant reduction in the amount of data leaving the drive to perform several common algorithms used in the Earth Science field of research.

The rest of the paper is outlined as follows. In section 8.1 we describe the design of the Active Drives and our software interface to them. Section 8.2 we briefly describe two common Earth Science algorithms that we implemented using MapReduce model on Active Drives. Section 8.3 we show the performance results from our experimental runs. Finally, we will provide a summary and conclusion in section 8.4.

8.1 Design

We used Seagate Kinetic Drives with modified software and firmware to allow for on drive computation. The Kinetic Drives are Ethernet-enabled object-store drives which use the open source Kinetic Protocol to execute drive operations.

Each Kinetic Drive has an extra ARM processor and an extra RAM chip on the PCBA which is connected to the outward facing Ethernet interface. This chip also runs a small Linux kernel which is capable of executing a full fledged application. The Kinetic Protocol was modified to allow for arbitrary program execution.

To integrate the Kinetic Drives with LVFS we implemented a new content module for LVFS capable of communicating with the drives using the Kinetic protocol. This module consisted of approximately 100 lines of code. We additionally developed a library to enable the MapReduce programming paradigm to be utilized on the drives. This library is responsible for distributing the map function to each drive, triggering its execution, and performing the tasks done by the MapReduce paradigm, such as sorting the output from the map function, grouping values together and passing the results to the reduce function.

8.2 Algorithms

To test the viability, we used several common real world algorithms used in the field of Earth Science. Our input data source was the Orbiting Carbon Observatory-2 (OCO-2), a NASA mission launched in 2014. The instrument measures column average CO_2 as it orbits the earth (Pollock *et al.* 2010; Crisp *et al.* 2004). Each input measurement consists of the Latitude, Longitude, measurement time, and CO_2 radiance value in addition to a quality flag indicating the quality of the measurement. The algorithms we ported to MapReduce on Active Drives are subsetting and gridding.

Subsetting is a reduction in data which involves only returning measurements that are of interest. Common ways to subset a dataset include spatial and temporal. As the name implies, spatial subsetting involves pruning the input to only include data within the bounding box of interest. Similarly, temporal subsetting involves pruning data for time periods of interests. Gridding involves converting measurements from an irregularly spaced input data to an output that consists of regularly spaced grid.

Subsetting is implemented as a straightforward scanning of the data. The pseudo code for subsetting is shown in Algorithm 3. Each measurement point is read in as input, the Lat/Lon coordinates are checked to ensure they fall within the bounding box, the measurement time is checked as the time range of interest, and finally the quality flag is checked to ensure the measurement is of good enough quality for use. If all checks pass, the input is written out as output repeating for the next line.

The gridding algorithm, shown as pseudo code in Algorithm 4, is implemented by computing a grid cell ID based on the Lat/Lon coordinates obtained for each measurement point of the input. For each grid cell we maintain two values; the first which is a summation of all measurements for this grid cell and a count of measurements which fell into this grid cell. Once all input has been processed, the summation and count values for each grid cell for which we computed values are returned as output. The output key is the grid cell ID and the output value is a concatenation of the summation and count values. The reduce algorithm receives all these key/value pairs from all map functions grouped by the key. For each grid cell ID, the summation and the counts are added up and the total summation value is divided by the total count. A more detailed look at this algorithm and comparisons non-Hadoop based ways of performing a simple gridding is explored in (Golpayegani & Halem 2009).

The MapReduce algorithm is converted to work with Active Drives by linking the reduce function into an executable to be run on the host compute node. In addition to the reduce function, the host compute node's executable consists of two managerial tasks. First, it is responsible for triggering the execution of map function on each Active Drive, second it is responsible for gathering the output of all map functions, sorting and grouping them by key and feeding the result to the reduce function.

The map function is compiled as an applet for the Active Drive and stored on each Active Drive ahead of the computation. The MapReduce task is triggered by the execution of the reduce executable on the host compute node. The first managerial task part of the reduce executable will identify all Active Drives which will be involved in the MapReduce computation and will trigger the startup of the Map applets on each drive. It will then monitor the applets for completion. After completion, the second managerial code will gather the output of all Map applets and feed them as input to the Reduce function. The output of the Reduce function is then written as the final result.

Algorithm 3 Spatial and Temporal subsetting

```
while INPUT do
    if Lat > LatLowerBound && Lat < LatUpperBound then
        if Lon > LonLowerBound && Lon < LonUpperBound then
            if time > StartTime && time < EndTime then
                if quality > AcceptableQualityValue then
                    print OUTPUT
                end if
            end if
        end if
    end if
end while
```

8.3 Performance

We measured two performance aspects of Hadoop and the Active drives; (1) the time it took to store data into HDFS or the Active drives and (2) the time to perform the subsetting and gridding operations. We used the Linux *time* utility to measure the real (wall clock) time for each of those aspects. On the Active drives we split the 2GB CSV file into 1MB

Algorithm 4 Gridding

```
while INPUT do
    if quality > AcceptableQualityValue then
        Compute grid cell from Lat/Lon;
        grid[grid cell] += measurement;
        gridCount[grid cell]++;
    end if
end while
for all gridcell ∈ grid do
    output grid[grid cell] / gridCount[grid cell];
end for
```

# of Nodes	Upload Time		Subsetting Time		Gridding Time	
	Hadoop	Active Drive	Hadoop	Active Drive	Hadoop	Active Drive
2	191	41	56	294	93	644
4	220	24	53	153	92	332
8	224	21	36	74	72	178
16	233	22	39	44	71	90
24	245	21	37	27	73	64

Table 8.1: Upload and Compute times for Hadoop and Active Drives in seconds.

blocks before uploading to the drives. On Hadoop, the entire 2GB CSV file was uploaded to HDFS and it was left to HDFS to perform the chunking and distribution to each DataNode.

Time measurements were taken using the Linux time utility and the real (wall clock) time was used for performance measurements. The maximum object size on the Active Drives is 1MB. The 2GB CSV file was split into 1MB blocks and uploaded to an Active Drive. For Hadoop, the 2GB CSV file was uploaded as a single file and it was left to Hadoop to perform the round robin chunking to each DataNode.

The runtimes are tabulated for easy reference in Table 8.1. For the subsetting experiment, the Active Drives performed computations on 2.4GB of data but only 3.1MB of data was communicated from the drives to the host. For Hadoop, all 2.4GB had to be read from each local drive to the host for computation and the same 3.1MB were then communicated from Hadoop back for retrieval.

Figure 8.1 shows the upload times between Hadoop and Active Drives between 2 and 24 nodes. We can see that the Active Drives are significantly faster than HDFS. It

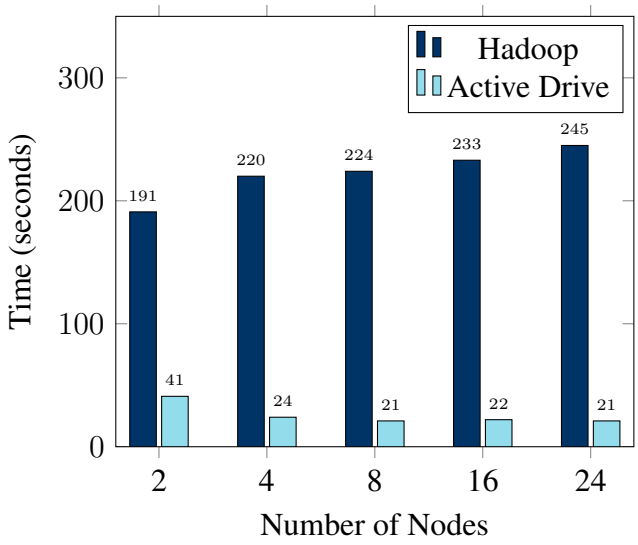


Figure 8.1: Upload times to Hadoop (HDFS) and Active Drives

takes approximately 41 seconds to upload 2GB of data to two Active Drives. This time reduces to approximately 21 seconds when using 24 drives. For HDFS the upload times are 191 seconds and 245 seconds for two and 24 HDFS nodes. We believe the increase in upload time in the Hadoop case is due to the additional network communication necessary between the DataNodes and the NameNode. As more nodes are added to the Hadoop, the NameNode becomes a bottleneck needing to coordinate with more DataNodes.

Figure 8.2 shows the time it took Hadoop and the Active Drives to perform subsetting using code described in algorithm 3. Hadoop's computation show slight improvements as the number of nodes is increased. The Active Drives, however, show significant improvements as their numbers are increased. As the number of drives are doubled the compute time is halved. At 24 nodes Hadoop performed the subsetting in 37 seconds while the Active Drives performed the same computation in 27 seconds.

Figure 8.3 shows the gridding times for the algorithm described in 4. Once again we can see some improvements from Hadoop between two and eight nodes. After eight nodes, however, there is no significant changes in the Hadoop performance for gridding. For the Active Drives, the compute time continues to halve between two and 24 nodes. At 24 nodes the Hadoop runtime is 73 seconds compared to the Active Drive's runtime of 64 seconds.

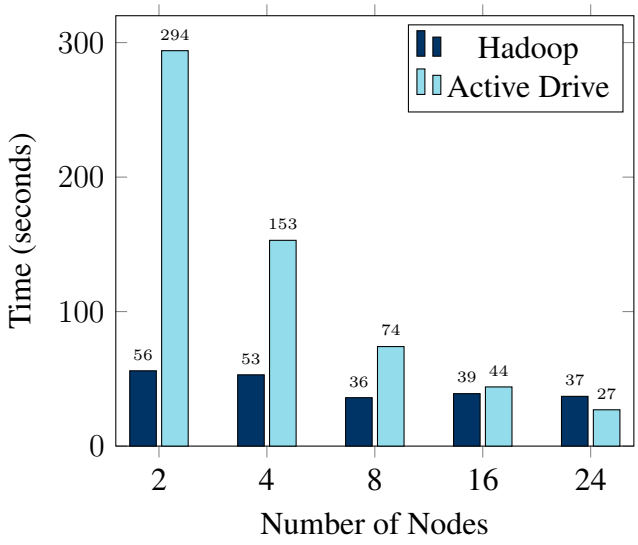


Figure 8.2: Subsetting times for Hadoop and Active Drives for different number of nodes

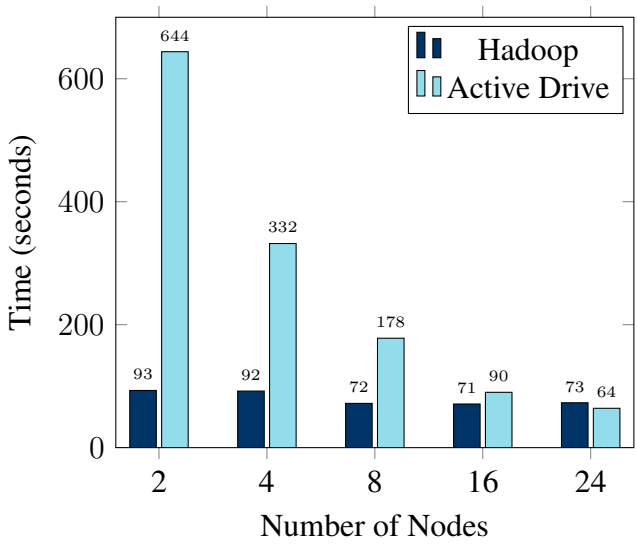


Figure 8.3: Gridding times for Hadoop and Active Drives for different number of nodes

8.4 Conclusion

Based on the results for this specific case study, we conclude that the Active Drive BlueSky chassis uploaded the data at 10 times the speed of Hadoop when comparing the best case numbers for each. For the same number of Hadoop and Active Drive nodes, the Active Drives were nearly 12 times faster for 24 nodes and 5 times faster for 2 nodes. The computation times for Active Drives and Hadoop were nearly identical for 24 nodes.

An additional benefit of the Active Drives is the reduction in the amount of data communicated out of the drives. In the case of the Hadoop, the entire 2GB of data was transferred to HDFS. During computation all 2GB of data were read back from HDFS, and hence the drives. With the Active Drives, after storing the 2GB, most of the data never left the drives. Only a few MB of data was transferred out of the drives after the on-drive map functions significantly reduced the data of interest. Additionally, more investigation is needed in explaining Hadoop's strange behavior in requiring more time to store data as the number of Hadoop nodes are increased.

We conclude based on these results that Active Drives are likely to be a good compliment to a compute system when there is significant subsetting/filtering to be achieved via simple computations. The low cost and low compute power on each drive would not be a complete replacement of a traditional compute system but rather an augmentation of it. The standard compute systems can perform the more complex and compute intensive tasks while offloading the easy to distribute and relatively easy compute tasks to the Active Drives.

Adopting a MapReduce environment appears to be a good methodology for achieving such a design with the Map functions performing the filtering and the reduce function performing the more complex tasks. Further research is needed to determine at what level of computational complexity the transition from Active Drive to regular computer is most beneficial.

Chapter 9

CONCLUSIONS

In this thesis, we developed the Lightweight Virtual File System, a reference implementation for a new file system concept in which a plugin framework controls the behavior and capabilities of the file system. We showed the ease with which such an implementation enables LVFS to merge older storage technologies with new ones. We tested LVFS with 3 use cases to show

which is flexible in dealing with evolving technologies and competing requirements between data centers and projects running in those data centers. We developed a plugin based file system which allowed for easy expansion of the file system capabilities and functionality. This plugin architecture enables LVFS to merge older storage technologies with newer ones without altering code accessing files on either storage technologies for the purpose of using both technologies or for smoothly transitioning to the new technology. We developed a YAML based configuration architecture designed for the flexibility to represent nearly any scenario of file system layout. We showed that even with the additional flexibility and overhead associated with this design, LVFS fully utilize the network hardware it runs on. LVFS was tested in 3 scenarios; (1) Data Distribution, (2) Data Migration, and (3) On-Drive MapReduce. With these case studies we validated our thesis statement showing our plugin framework combined with the configuration design of LVFS we are capable of creating a flexible and fully functional file system with contributions which include: (1) A plugin framework, (2) Metadata Replacement, (3) Dynamic views, (4) On-Drive MapReduce, (5) Platform agnostic file content, and (6) Configuration.

Plugin Framework

We designed a plugin framework capable a full featured file system. LVFS assigned all required POSIX functions for a full featured file system to different plugin cat-

egories. LVFS core code acted as the layer coordinating communication between these plugins to create a file system.

Metadata Replacement

We successfully removed the metadata component of a file system and relied entirely on metadata provided by the user. Using the plugin framework we successfully designed a metadata plugin category, which can obtain all necessary information from any metadata source, provided a plugin has been developed for it.

Dynamic views

We successfully showed the ability to dynamically create directory structures from the metadata plugins. We were able to easily change directory structures by changing the LVFS configuration file.

On-Drive MapReduce

We successfully developed a framework for performing On-Drive MapReduce. We developed a plugin which was capable of storing on state of the art active drives and were able to design a generic MapReduce framework for the drives without requiring users to write drive specific code.

Platform agnostic file content

We showed LVFS capabilities to retrieve and store files from incompatible sources in a unified layout. New content plugins create the ability for LVFS to handle new sources and treat all sources equally for the file system users.

Configuration

We showed LVFS' ability to change capabilities with its configuration design. With changes in configuration, LVFS can add new plugins which add new functionality to LVFS.

9.1 Future Work

For quick development LVFS utilized the Fuse library. The use of Fuse allowed for fast code deployment since no kernel modifications were necessary. This, however, puts LVFS at a performance handicap compared to other file systems. To remove this handicap

we plan on moving LVFS into the Linux kernel once most LVFS features have matured. We believe this change will have significant performance improvements in LVFS particularly when small block sizes are used.

LVFS caching strategies is based on a flat caching structure. Files are stored in a single cache module and retrieved out of it. We plan on exploring a hierarchical caching strategy in which multiple layers of cache are utilized for better performance. LVFS could use small but very fast Solid State Disks for more frequently used files while using bigger but slower hard drives for less frequent files.

LVFS needs to develop better features for on-drive computation. We have shown the viability of performing on-drive computation using MapReduce. However, vanilla MapReduce computation is not suitable for all types of computations. For example, algorithms that require iteration are not suitable for standard MapReduce algorithms. Expanding LVFS to better support on-drive computation and support other distributed compute modules for on-drive computation will expand the class of algorithms capable of using LVFS.

In addition to having a hierarchical cache structure, LVFS needs to support a tier based storage architecture. With such a system LVFS could move less frequently used files to slower but more abundant storage and keep more frequently used files on faster but more expensive files. With such a system LVFS could utilize a tier based system ranging from tape storage to Non-volatile random access memory for very fast access and high amount of storage capacity.

We showed the ease with which LVFS is capable of transitioning between storage architectures. However, some of the necessary steps for transitioning require manual interventions, such as when copying files from old storage architecture to new. A good contribution would be to develop a configuration design inside LVFS' existing configuration structure such that this copying can be performed automatically with proper failure detection.

REFERENCES

[Acharya, Uysal, & Saltz 1998] Acharya, A.; Uysal, M.; and Saltz, J. 1998. Active disks: Programming model, algorithms and evaluation. *ACM SIGPLAN Notices* 33(11):81–91.

[Amazon Web Services, Inc. 2013] Amazon Web Services, Inc. 2013. Amazon S3, Cloud Computing Storage for Files, Images, Videos. [Online; accessed 09-Oct-2013].

[Azar *et al.* 1999] Azar, Y.; Broder, A. Z.; Karlin, A. R.; and Upfal, E. 1999. Balanced allocations. *SIAM journal on computing* 29(1):180–200.

[Ben-Kiki, Evans, & Ingerson 2005] Ben-Kiki, O.; Evans, C.; and Ingerson, B. 2005. Yaml ain't markup language (yaml) version 1.1. *yaml. org, Tech. Rep*.

[Berners-Lee, Fielding, & Masinter 1998] Berners-Lee, T.; Fielding, R.; and Masinter, L. 1998. Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396, RFC Editor.

[Berners-Lee, Fielding, & Masinter 2005] Berners-Lee, T.; Fielding, R.; and Masinter, L. 2005. Uniform Resource Identifiers (URI): Generic Syntax. RFC 3986, RFC Editor.

[bji 2017] bji. 2017. libs3. [Online; accessed 1-Jun-2017].

[Borthakur 2008] Borthakur, D. 2008. HDFS architecture guide. [Online; accessed 14-Jul-2015].

[Brumfiel 2011] Brumfiel, G. 2011. Down The Petabyte Highway. *Nature* 469(20):282–283.

[Chu *et al.* 2006] Chu, C.-T.; Kim, S. K.; Lin, Y.-A.; Yu, Y.; Bradski, G.; Ng, A. Y.; and Olukotun, K. 2006. Map-reduce for machine learning on multicore. In *NIPS*, volume 6, 281–288. Vancouver, BC.

[Crisp *et al.* 2004] Crisp, D.; Atlas, R.; Breon, F.-M.; Brown, L.; Burrows, J.; Ciais, P.; Connor, B.; Doney, S.; Fung, I.; Jacob, D.; et al. 2004. The orbiting carbon observatory (oco) mission. *Advances in Space Research* 34(4):700–709.

[Dabek *et al.* 2002] Dabek, F.; Zeldovich, N.; Kaashoek, F.; Mazières, D.; and Morris, R. 2002. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, 186–189. ACM.

[DataDirect Networks 2017] DataDirect Networks. 2017. WOS OBJECT STORAGE. `http://www.ddn.com/download/resource_library/brochures/object_storage/ddn-wos-objectstorage-brochure_2.pdf`.

[Dean & Ghemawat 2004] Dean, J., and Ghemawat, S. 2004. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, 10–10. Berkeley, CA, USA: USENIX Association.

[DeCandia *et al.* 2007] DeCandia, G.; Hastorun, D.; Jampani, M.; Kakulapati, G.; Lakshman, A.; Pilchin, A.; Sivasubramanian, S.; Vosshall, P.; and Vogels, W. 2007. Dynamo: Amazon's highly available key-value store. In *SOSP*, volume 7, 205–220.

[Delmerico *et al.* 2009] Delmerico, J. A.; Byrnes, N. A.; Bruno, A. E.; Jones, M. D.; Gallo, S. M.; and Chaudhary, V. 2009. Comparing the performance of clusters, Hadoop, and Active Disks on microarray correlation computations. In *High Performance Computing (HiPC), 2009 International Conference on*, 378–387. IEEE.

[Folk *et al.* 2011] Folk, M.; Heber, G.; Koziol, Q.; Pourmal, E.; and Robinson, D. 2011. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, 36–47. ACM.

[Forster 2017] Forster, F. 2017. collectd The system statistics collection daemon. `https://collectd.org/`.

[Fryer *et al.* 2012] Fryer, D.; Sun, K.; Mahmood, R.; Cheng, T.; Benjamin, S.; Goel, A.; and Brown, A. D. 2012. Recon: Verifying File System Consistency At Runtime. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, 7–7. USENIX Association.

[Gantz & Reinsel 2012] Gantz, J., and Reinsel, D. 2012. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the Future*.

[Golpayegani & Halem 2009] Golpayegani, N., and Halem, M. 2009. Cloud computing for satellite data processing on high end compute clusters. In *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*, 88–92. IEEE.

[Grafana Labs 2017] Grafana Labs. 2017. The open platform for beautiful analytics and monitoring. `https://grafana.com/`.

[Group 2014] Group, T. O. 2014. DCE – OpenDCE – Portal. `http://www.opengroup.org/dce/`.

[Grünbacher 2003] Grünbacher, A. 2003. POSIX Access Control Lists on Linux. In *USENIX Annual Technical Conference, FREENIX Track*, 259–272.

[Han *et al.* 2011] Han, J.; Haihong, E.; Le, G.; and Du, J. 2011. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, 363–366. IEEE.

[Honicky & Miller 2004] Honicky, R., and Miller, E. L. 2004. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 96. IEEE.

[Hua *et al.* 2009] Hua, Y.; Jiang, H.; Zhu, Y.; Feng, D.; and Tian, L. 2009. SmartStore: A New Metadata Organization Paradigm with Semantic-Awareness for Next-Generation File Systems. In *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, 1–12. IEEE.

[Ierusalimschy, De Figueiredo, & Celes Filho 1996] Ierusalimschy, R.; De Figueiredo, L. H.; and Celes Filho, W. 1996. Lua-an extensible extension language. *Softw., Pract. Exper.* 26(6):635–652.

[Inc. 2017] Inc., M. 2017. Mongodb for giant ideas. [Online; accessed 9-May-2017].

[Jeger & Pautasso 2008] Jeger, M. J., and Pautasso, M. 2008. Plant disease and global change–the importance of long-term data sets. *New Phytologist* 177(1):8–11.

[Jones 2009] Jones, M. 2009. Anatomy of the Linux virtual file system switch. https://www.ibm.com/developerworks/library/l-virtual-filesystem-switch/. [Online; Accessed 5-May-2017].

[Karlsson *et al.* 2001] Karlsson, J. S.; Pham, T.; Lal, A.; and Leung, C. 2001. Ibm db2 everyplace: A small footprint relational database system. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 0230–0230. IEEE Computer Society.

[Keeton, Patterson, & Hellerstein 1998] Keeton, K.; Patterson, D.; and Hellerstein, J. 1998. The case for intelligent disks (idisks). In *Sigmod Record*, 27(3):42–52.

[Labs 2017] Labs, R. 2017. Redis. [Online; accessed 9-May-2017].

[Lakshman & Malik 2010] Lakshman, A., and Malik, P. 2010. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review* 44(2):35–40.

[Leavitt 2010] Leavitt, N. 2010. Will nosql databases live up to their promise? *Computer* 43(2).

[Leung *et al.* 2009] Leung, A. W.; Shao, M.; Bisson, T.; Pasupathy, S.; and Miller, E. L. 2009. Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems. In *Proccedings of the 7th conference on File and storage technologies*, 153–166. USENIX Association.

[libfuse 2014] libfuse. 2014. FUSE: Filesystem in Userspace. [Online; Accessed 5-May-2017].

[Long & Miller 2015] Long, D. E., and Miller, E. L. 2015. Dynamic Non-Hierarchical File Systems for Exascale Storage. Technical report, Univ. of California, Santa Cruz, CA (United States).

[Massie, Chun, & Culler 2004] Massie, M. L.; Chun, B. N.; and Culler, D. E. 2004. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing* 30(7):817–840.

[Mesnier, Ganger, & Riedel 2003] Mesnier, M.; Ganger, G. R.; and Riedel, E. 2003. Object-Based Storage. *Communications Magazine, IEEE* 41(8):84–90.

[Naps, Mokbel, & Du 2011] Naps, J. L.; Mokbel, M. F.; and Du, D. H. 2011. Pantheon: Exascale File System Search for Scientific Computing. In *Scientific and Statistical Database Management*, 461–469. Springer.

[National Oceanographic and Atmospheric Administration 2014] National Oceanographic and Atmospheric Administration. 2014. NEXRAD Data Inventory Search — National Climatic Data Center . [Online; accessed 06-Oct-2014].

[Owens & Allen 2010] Owens, M., and Allen, G. 2010. *SQLite*. Springer.

[Papadimitriou *et al.* 2000] Papadimitriou, C. H.; Tamaki, H.; Raghavan, P.; and Vempala, S. 2000. Latent Semantic Indexing: A Probabilistic Analysis. *Journal of Computer and System Sciences* 61(2):217–235.

[Pollock *et al.* 2010] Pollock, R.; Haring, R. E.; Holden, J. R.; Johnson, D. L.; Kapitanoff, A.; Mohlman, D.; Phillips, C.; Randall, D.; Rechsteiner, D.; Rivera, J.; et al. 2010. The orbiting carbon observatory nstrument: performance of the oco instrument and plans for the oco-2 instrument. In *Remote Sensing*, 78260W–78260W. International Society for Optics and Photonics.

[Rajasekar *et al.* 2010] Rajasekar, A.; Moore, R.; Hou, C.-y.; Lee, C. A.; Marciano, R.; de Torcy, A.; Wan, M.; Schroeder, W.; Chen, S.-Y.; Gilbert, L.; et al. 2010. iRODS Primer: Integrated Rule-Oriented Data System. *Synthesis Lectures on Information Concepts, Retrieval, and Services* 2(1):1–143.

[Riedel *et al.* 2001] Riedel, E.; Faloutsos, C.; Gibson, G. A.; and Nagle, D. 2001. Active disks for large-scale data processing. *Computer* 34(6):68–74.

[Riedel, Gibson, & Faloutsos 1998] Riedel, E.; Gibson, G. A.; and Faloutsos, C. 1998. Active storage for large-scale data mining and multimedia. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, 62–73. Morgan Kaufmann Publishers Inc.

[Riedel 1999] Riedel, E. 1999. *Active Disks Remote Execution for Network-Attached Storage*. Ph.D. Dissertation, Carnegie Mellon University, Pittsburgh, PA.

[Ritter *et al.* 2000] Ritter, N.; Ruth, M.; Grissom, B. B.; Galang, G.; Haller, J.; Stephenson, G.; Covington, S.; Nagy, T.; Moyers, J.; Stickley, J.; et al. 2000. Geotiff format specification geotiff revision 1.0. *SPOT Image Corp*.

[Ross & Thakur 2000] Ross, R. B., and Thakur, R. 2000. PVFS: A Parallel File System for Linux Clusters. In *in Proceedings of the 4th Annual Linux Showcase and Conference*, 391–430.

[Schwan 2003] Schwan, P. 2003. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium*, volume 2003.

[Seagate Technology LLC 2017] Seagate Technology LLC. 2017. The Seagate Kinetic Open Storage Vision. [Online; accessed 27-Apr-2017].

[Shepler *et al.* 2003] Shepler, S.; Eisler, M.; Robinson, D.; Callaghan, B.; Thurlow, R.; Noveck, D.; and Beame, C. 2003. Network file system (NFS) version 4 protocol. *Network*.

[Shreedhar & Varghese 1995] Shreedhar, M., and Varghese, G. 1995. Efficient fair queueing using deficit round robin. *ACM SIGCOMM Computer Communication Review* 25(4):231–242.

[Shvachko 2010] Shvachko, K. V. 2010. HDFS Scalability: The limits to growth. *;login* 35(2):6–16.

[Stone & Partridge 2000] Stone, J., and Partridge, C. 2000. When the CRC and TCP Checksum Disagree. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, 309–319. New York, NY, USA: ACM.

[Szalay & Gray 2006] Szalay, A., and Gray, J. 2006. 2020 computing: Science in an exponential world. *Nature* 440(7083):413–414.

[The Economist 2014] The Economist. 2014. Data, data everywhere. [Online; accessed 14-Jul-2014].

[The Open Group 2016a] The Open Group. 2016a. dlclose. [Online; accessed 17-May-2017].

[The Open Group 2016b] The Open Group. 2016b. dlopen. [Online; accessed 17-May-2017].

[The Open Group 2017] The Open Group. 2017. The Open Group Base Specifications Issue 7. http://pubs.opengroup.org/onlinepubs/9699919799/toc.htm.

[The PostgreSQL Global Development Group 2017] The PostgreSQL Global Development Group. 2017. What is PostgreSQL? https://www.postgresql.org/docs/current/static/intro-whatis.html.

[Tian *et al.* 2012] Tian, Y.; Klasky, S.; Yu, W.; Abbasi, H.; Wang, B.; Podhorszki, N.; Grout, R.; and Wolf, M. 2012. SMART-IO: SysteM-AwaRe Two-Level Data Organization for Efficient Scientific Analytics. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, 181–188. IEEE.

[Tommasi *et al.* 2017] Tommasi, D.; Stock, C. A.; Hobday, A. J.; Methot, R.; Kaplan, I. C.; Eveson, J. P.; Holsman, K.; Miller, T. J.; Gaichas, S.; Gehlen, M.; et al. 2017. Managing living marine resources in a dynamic environment: the role of seasonal to decadal climate forecasts. *Progress in Oceanography*.

[Vilayannur *et al.* 2002] Vilayannur, M.; Ross, R. B.; Carns, P. H.; Thakur, R.; Sivasubramaniam, A.; and Kandemir, M. 2002. Improving the performance of the posix i/o interface to pvfs.

[Watson & Coyne 1995] Watson, R. W., and Coyne, R. A. 1995. The parallel I/O architecture of the high-performance storage system (HPSS). In *Mass Storage Systems, 1995.'Storage-At the Forefront of Information Infrastructures', Proceedings of the Fourteenth IEEE Symposium on*, 27–44. IEEE.

[Western Digital Corporation 2017] Western Digital Corporation. 2017. HGST Active Archive System. https://www.hgst.com/sites/default/files/resources/ActiveArchive-OSS-DS.pdf.

[White 2009] White, T. 2009. *Hadoop: The Definitive Guide*. "O'Reilly Media, Inc.".

[Wikipedia 2017] Wikipedia. 2017. Filesystem in userspace. [Online; accessed 5-May-2017].

[Zhao & Pjesivac-Grbovic 2009] Zhao, J., and Pjesivac-Grbovic, J. 2009. Mapreduce: The programming model and practice.